AD-A073 454 LOGICON INC SAN PEDRO CALIF F/G 9/2

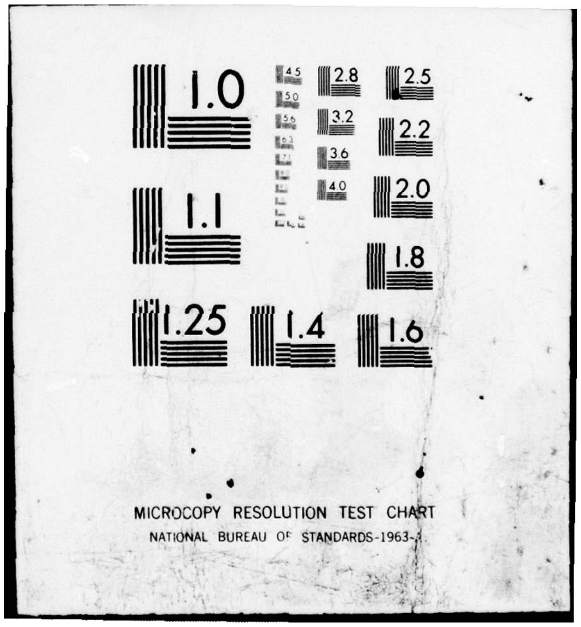
RELIABLE MICROPROGRAMMING.(U)

JUL 79 J RADATZ, J LAUB, G HAMACHI F30602-78-C-0079

UNCLASSIFIED DS-R79001 RADC-TR-79-173 NL

1 OF 4

AD
A073454


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-79-173
Final Technical Report
July 1979

RELIABLE MICROPROGRAMMING

Logicon, Inc.

Jane Radatz
Jeffrey Laub
Gordon Hamachi

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-79-173 has been reviewed and is approved for publication.

APPROVED: [illegible]

DONALD F. ROBERTS
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: [illegible]

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 **REPORT DOCUMENTATION PAGE**		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-79-173	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 RELIABLE MICROPROGRAMMING		5. TYPE OF REPORT & PERIOD COVERED 9 Final Technical Report, 7 Feb 78 — 6 Feb 79
		6. PERFORMING ORG. REPORT NUMBER 14 DS-R79001
7. AUTHOR(s) 10 Jane Radatz, Jeffrey Laub, Gordon Hamachi		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-78-C-0079
9. PERFORMING ORGANIZATION NAME AND ADDRESS Logicon, Inc. 255 West 5th Street San Pedro CA 90731		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 16 25310202 17 02
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE 11 July 1979
		13. NUMBER OF PAGES 351 12 347 p.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald Roberts (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microprogramming
verification and validation
software tools
microprogramming tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the results of the Reliable Microprogramming Contract. Current technology for microprogram verification is surveyed, and a set of microprogramming tools is proposed. These tools are evaluated, and a recommended set is identified. A Functional Description and a System Specification are provided for each recommended tool.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 061

JCB

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

Evaluation

The objective of this effort was to develop techniques and define automated software tools for the verification and validation of microprograms. Applicable technology areas included: source code static analysis, dynamic analysis, on-line debugging and programming procedures. Functional Descriptions and System/Subsystem Specifications per DoD 4120.17-M were to be developed for each tool identified. The effort is responsive to RADC TPO-R5A, Software Cost Reduction.

The effort resulted in the identification of eleven candidate tools for microprogram verification and validation. Functional Descriptions and System/Subsystem Specifications were developed for eight of the eleven tools identified.

Although the original intent of the effort was to define tools for microprograms, the specifications produced are applicable to any programming environment. The individual specifications serve as a useful basis for implementing individual tools to satisfy limited requirements, or taken in total, define a comprehensive software testing environment. The documentation will be used to specify testing requirements for various programming environments in future developments.

[illegible]
Donald F. Roberts
Project Engineer

SUMMARY

The objective of the Reliable Microprogramming Contract was to define a set of automated tools and techniques that could be used for microprogram verification and validation at the Rome Air Development Center (RADC). The first part of the project was devoted to a study of current technology and requirements and to the proposal and evaluation of new microprogramming tools. The approach taken may be summarized as follows:

- o Review of current literature on software and firmware tools
- o Identification of problem areas in microcode development
- o Evaluation of the ability of current tools to address the problem areas
- o Postulation of additional tools to deal with the unresolved problem areas
- o Definition and application of a tool selection metric
- o Identification of a recommended set of microprogramming tools

The following tools were recommended, in order of preference:

- o Microcode Comparison Program
- o Global Cross-Reference Generator
- o Programming Practices Monitor
- o Control Flow Analyzer
- o Timing Analyzer
- o Data Flow Analyzer
- o Execution Monitor
- o Symbolic Executor
- o Test Case Generator

The second part of the project focused on the development of functional specifications and usage scenarios for these tools. With the exception of the Programming Practices Monitor, for which it was determined that specification was premature, the requirements for each tool were identified and set forth in both a Functional Description and a System Specification. These specifications are designed to serve as the basis for development of the recommended tools.

PREFACE

This report presents the results of the Reliable Microprogramming Project performed by Logicon, Inc., under Contract F30603-78-C-0079 with the Rome Air Development Center, Air Force Systems Command. The work was performed during the period 7 February 1978 to 6 February 1979. The authors of this document are Jane Radatz, Jeffrey Laub, and Gordon Hamachi. George Cannon, Paul Courcy, Anita Friedman, David Weeton, and Mary Yee also contributed. Technical direction was provided by Mr. Donald Roberts, the Project Officer.

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION

This document is the final report for the Reliable Microprogramming Contract. The objective of the contract was to define a set of automated tools and techniques to support microprogram verification and validation at the Rome Air Development Center (RADC).

The use of tools to support software development is a well established technology. The emergence of higher order languages has greatly facilitated software coding, debugging, and maintenance, and a large variety of development and testing tools are available to aid in the software development process.

Microprogramming technology can be compared with software technology as it existed 20 years ago. It is a tedious process performed by highly skilled individuals with little or no support from automated tools. The technology for building higher order language compilers to generate microcode is in its infancy. Debugging and testing techniques are crude. Few analysis or testing tools have emerged.

The problem is magnified by the fact that microprogramming is inherently more difficult than standard software coding. Microprogrammers must have a thorough understanding of the processor at the machine architecture level. Timing is critical because of the possibility that a single microinstruction may perform concurrent operations. Particular attention must be paid to control data flow and allocation of machine resources to avoid destroying intermediate results.

If the application is the emulation of another computer, the difficulty is compounded. The same detailed knowledge of the emulated machine is required and, in addition, optimization of the microcode is critical to achieving an emulation that executes in real time or better.

The Reliable Microprogramming Contract was performed in response to these concerns. The contract was undertaken to determine the extent to which the technology of software tools could be transferred to the microprogramming environment.

RADC has recently acquired a Nanodata QM-1 microprogrammable computer that will serve as the nucleus of a System Architecture Evaluation Facility. This facility will be an integrated collection of hardware and software tools for emulating and evaluating a wide range of computer system architectures. The QM-1 will be interfaced to the H6180/MULTICS or DEC System 20 and will be available over the ARPANET.

In a microprogrammed computer, each machine instruction points to an address in a fast "control store." The microprogram at that address provides control signals to implement the machine instruction. In the QM-1, there are actually two levels of microprogrammed control. Just as machine instructions are executed by microprograms in control store, microinstructions are executed by nanoprograms in a "nanostore."

Microinstructions and nanoinstructions in the QM-1 differ in both format and nature. Nanoinstructions are 360 bits wide, each bit determining the control signal to a specific hardware component of the system. This format provides extreme flexibility and permits specification of parallel operation of hardware components. Microinstructions are only 18 bits wide and resemble machine instructions in that they have an op-code and several other encoded fields. This format provides a limited selection of control patterns and typically specifies a sequence of control signals rather than a set of parallel operations.

Current microprogramming at RADC is directed almost solely to the microcode level. As a result, Logicon was directed to focus on this level in the Reliable Microprogramming Contract. The findings and recommendations presented in this report therefore concentrate on microcode verification and validation and do not address the issues that arise in the development of nanocode.

The project was divided into two parts. The first part, described in Section 2, consisted of the review and assessment of current technology and requirements and the identification of a recommended set of tools for microprogram verification and validation. The second part, described in Section 3, focused on the development of functional specifications and usage scenarios for the recommended tools. Conclusions and recommendations for the project are presented in Section 4. Appendixes A through F contain supporting data; Appendixes G through Z provide Functional Descriptions and System Specifications for the recommended microprogramming tools.

2. THE RELIABLE MICROPROGRAMMING STUDY

The first part of the Reliable Microprogramming Contract consisted of a study of microprogram verification and validation technology. This section describes the approach taken and presents the results for the 10 tasks performed:

1) Identification of existing software tools
2) Identification of existing microprogramming tools
3) Identification of problem areas in microcode development
4) Evaluation of the applicability of current microprogramming tools to the microprogramming problem areas
5) Evaluation of the applicability of current software tool concepts to the microprogramming problem areas
6) Identification of unresolved problem areas
7) Postulation of new microprogramming tools
8) Assessment of the impact of the proposed microprogramming tools on the microprogramming problem areas
9) Development of a tool selection metric
10) Application of the tool selection metric

2.1 Identification of Existing Software Tools

Task 1 identified state-of-the-art tools and techniques used for verifying and validating higher-order and assembly language software. The approach taken was to review available literature and develop a set of abstracts summarizing the capabilities, limitations, and other relevant characteristics of representative current tools. Table 1 identifies the tools selected. Appendix A defines the tool classifications used in the table and provides the tool abstracts.

2.2 Identification of Existing Microprogramming Tools

Task 2 identified state-of-the-art tools and techniques used for the verification and validation of microprograms. The approach taken was the same as that for Task 1, namely, to review available literature and develop a set of tool abstracts. As expected, much less information was available on microprogramming tools than on software tools, and a smaller group of tools was selected. Table 2 identifies these tools. Appendix B contains the tool abstracts.

2.3 Identification of Problem Areas in Microcode Development

The purpose of Task 3 was to identify a set of microprogramming problem areas that could serve as a basis for determining the utility of current and proposed microprogram verification and validation tools. The initial approach was to collect and then group specific problems encountered by microcode development personnel. This approch was not successful because:

Table 1. Existing Software Tools

Name	Classification
ACES (Automated Code Evaluation System)	Cross-reference generator; execution monitor
AIDS	Abort diagnosis tool; breakpoint controller; execution monitor
AMPIC (Automatic Machine Processor Inference Confirmer)	Breakpoint controller; cross-reference generator; flowchart generator; simulator/symbolic executor
ATDG (Automated Test Data Generator)	Test case generator
AUT (Automated Unit Test)	Simulator/symbolic executor
BAP (Branch Analysis Program)	Execution monitor
CADSAT (Computer-Aided Design and Specification Analysis Tool)	Requirements analysis aid
CLARKE (Clarke's Program Testing System)	Test case generator; formal verifier
CWRU (Case Western Reserve University Program Verification System)	Formal verifier
DAVE	Program auditor
DISSECT	Simulator/symbolic executor
EFFIGY	Breakpoint controller; simulator/symbolic executor; formal verifier
FACES (FORTRAN Automated Code Evaluation System)	Program auditor
FADEBUG-1	Simulator/symbolic executor
PET (Program Evaluator and Tester)	Execution monitor
SDVS (Software Design and Verification System)	File manager; simulator/symbolic executor
STRUCT (Structured Programming Verifier)	Program auditor
TPL (Test Procedure Language)	Simulator/symbolic executor

Table 2. Existing Microprogramming Tools

Name	Classification
BFTSC ICS (Brassboard Fault-Tolerant Spaceborne Computer Microcode Interpretive Computer Simulation)	Abort diagnosis tool; breakpoint controller
Interactive Debugger	Abort diagnosis tool; breakpoint controller
MCS (Microprogram Certification System)	Simulator/symbolic executor; formal verifier
PRIM (Programming Research Instrument) Debugger	Abort diagnosis tool; breakpoint controller
STRUM (Structured Microprogramming Language)	Formal verifier

- Documentation on microcode development problems is scarce.

- The information that was available was described in very machine-specific terms.

In the hopes that software development problem areas might also apply to microcode development, Logicon verification and validation reports were reviewed and a tentative set of 12 problem areas was established. These problem areas, defined in Table 3, were discussed with Logicon microcode developers and verifiers, found to be representative of microcode development problems, and selected for the study. Application of these problem areas to subsequent tasks showed them to be highly effective.

2.4 Evaluation of the Applicability of Current Microprogramming Tools to the Microprogramming Problem Areas

Task 4 estimated the degree to which the microprogramming tools identified in Task 2 apply to the microprogramming problem areas identified in Task 3. Table 4 presents the results of this evaluation. The criteria used to assign ratings to the tools were as follows:

- High: The tool applies directly to the problem area and provides highly significant and useful results

- Medium: The tool applies to the problem area, but provides information that is somewhat limited in scope, significance, or usefulness

- Low: The tool addresses the problem area only incidentally and provides information of very limited scope, significance, and usefulness

- Blank: The tool does not apply to the problem area at all

As Table 4 shows, seven of the twelve problem areas are well covered by existing microprogramming tools, having had one or more tools of high applicability assigned to them. Two problem areas, namely, errors in timing and process allocation and errors in interruptibility and data coherency, have tools of at best medium applicability. Two others, incomplete or erroneous specifications and incomplete or erroneous documentation, have tools of only low applicability. One problem area, violation of programming practices, is not addressed at all by the current microprogramming tools.

2.5 Evaluation of the Applicability of Current Software Tool Concepts to the Microprogramming Problem Areas

The task of evaluating the applicability of current software tool concepts to the microprogramming problem areas was originally conceived of as two separate tasks, one focusing on problem areas that are addressed by current microprogramming tools and one on areas that are not. Since Task 4

Table 3. Microprogramming Problems Areas

1. Incomplete or Erroneous Specifications: Errors of omission and commission in requirements or design specifications

2. Violation of Programming Practices: Errors caused by failure to comply with a prescribed set of programming standards, including syntactic errors caught by conventional compilers and assemblers but primarily oriented toward rules and conventions not automatically identified by these tools

3. Disagreement Between Code and Detailed Design Specification: Errors relating to missing code, extra code, or code that disagrees with the detailed design specifications, flow diagrams, HIPOs, or program descriptions

4. Incorrect Accessing or Storing of Data: Errors related to addressing, code modification, and scope of variabes

5. Errors in Equation Computation or Arithmetic: Errors related to scaling, rounding, precision, overflow, and underflow

6. Branching Errors: Errors related to branching to the wrong location because of an error in the test setup or in the branch test itself

7. Incorrect Constants or Data Formats: Errors related to improper transcription, format, or units conversion of program data

8. Errors in Timing and Process Allocation: Errors related to violation of timing constraints, particularly those associated with the timing relationships between concurrent operations

9. Errors in Interruptibility and Data Coherency: Errors related to reentrancy, enabling and disabling of interrupts, and timing relationships modified by the occurrence of interrupts

10. Logic and Sequence Errors: Errors related to code that differs from the detailed design specification in the order in which the various program steps are performed (e.g., caused by a spurious attempt at code optimization) (this type of error is a subset of Category 3 but was deemed sufficiently important to warrant separate identity)

11. Erroneous Use of System Hardware/Software/Firmware: Errors related to improper usage of (or interface to) any other system element (hardware, software, or firmware)

12. Incomplete or Erroneous Documentation: Errors or inadequacies related to product documentation such as user's manuals, maintenance manuals, C5 specifications, etc.

Table 4. Applicability of Existing Microprogramming Tools to Microprogramming Problem Areas

	Problem Area											
Tool	1	2	3	4	5	6	7	8	9	10	11	12
BFTSC ICS			Low	Low	Low	Med	Low	Med	Med		Med	
Interactive Debugger			Low	Low	Low	Low	Low	Med	Med		High	
MCS	Low		High	High	High	High	High		Low	High	Low	Low
PRIM Debugger			Low	Low	Low	Low	Low	Med	Med		High	
STRUM	Low		High	High	High	High	High		Low	High	Low	Low
Highest rating	Low		High	High	High	High	High	Med	Med	High	High	Low

had established that all problem areas except one were addressed at least minimally by existing microprogramming tools, it was decided to combine these two tasks into a single task for all of the problem areas.

The evaluation of each software tool entailed answering three questions:

- o Does the tool's concept apply to microprogram verification and validation as well as to software verification and validation?

- o How adequately would a microprogramming tool incorporating the capabilities of the software tool address each microprogramming problem area?

- o Would such a microprogramming tool offer significant improvement over the capabilities of existing microprogramming tools?

The results of the evaluation are summarized in Table 5. The ratings "high," "medium," and "low" reflect a combined response to the first two questions. They indicate that a microprogramming tool incorporating the software tool's capabilities:

- o High: Is feasible and is highly applicable to the problem area

- o Medium: Is feasible but has somewhat limited applicability to the problem area

- o Low: Is feasible but has very limited applicability to the problem area

- o Blank: Is not feasible, or is not applicable to the problem area

The asterisks in the table represent a positive response to the third question: a microprogramming tool incorporating the capabilities of the given software tool would offer a significant improvement over existing microprogramming tools. Appendix C contains supporting text for the table.

A number of observations may be made from Table 5. As the last line of the table shows, six of the problem areas have assigned to them tools of high applicability and the other six have tools of medium applicability, indicating that software tool concepts are well adapted to microprogramming problems. Furthermore, the asterisks indicate that in eight of the problem areas microprogramming tools incorporating the ideas of software tools would offer significant improvement over existing microprogramming tools. The conclusions reached were as follows:

Table 5. Applicability of Software Tool Concepts to Microprogramming Problem Areas

Tool	Problem Area 1	2	3	4	5	6	7	8	9	10	11	12
ACES			Med	Med*	Med*	Med				Med		Low
AIDS			Low	Med	Low	Med	Low		Med	Med	Med	Low
AMPIC			High*	High	High*	High*		Med*		High		Med*
ATDG			Med			Med				Med		
AUT			Low								Low	
BAP				Med*	Med*	High				Med		
CADSAT	Med*											
CLARKE			High	Med	High	High				High		
CWRU	Low		High	High	High	High	High		Low	High	Low	Low
DAVE			Med							Low		Med*
DISSECT			High	Med	High	High				High		Med
EFFIGY			High	Med	High	High				High		Med
FACES		Low		Med*								Med
FADEBUG-1			Low								Low	
PET			Low	High*	Med*	High		Med		Med		
SDVS			Low								Low	
STRUCT		Med*										
TPL/F			Low								Low	
Highest rating	Med*	Med*	High*	High*	High*	High*	High	Med*	Med	High	Med	Med*

*Significant improvement over existing microprogramming tools

- An important source of new microprogramming tools should be the ideas implemented in existing software tools.
- Tools similar to AMPIC, BAP, CLARKE, CWRU, DISSECT, EFFIGY, and PET would contribute significantly to microprogram verification and validation.
- Tools similar to AIDS, CADSAT, DAVE, FACES, and STRUCT would offer aid in problem areas having no highly applicable tools.

2.6 Identification of Unresolved Problem Areas

Task 6 identified microprogramming problem areas not adequately addressed by either existing microprogramming tools or adaptable software tools. Two independent methods were used to accomplish this task.

The first method was a straightforward interpretation of the results given in Tables 4 and 5. Seven of the twelve problem areas have tools of high applicability in one or both tables. These areas were considered to be adequately addressed. The other five areas have tools of only medium applicability and were therefore considered to be unresolved. These unresolved areas were as follows:

1) Incomplete or erroneous specifications
2) Violation of programming practices
8) Errors in timing and process allocation
9) Errors in interruptibility and data coherency
12) Incomplete or erroneous documentation

The second method, undertaken to verify these conclusions, entailed grouping the capabilities of each microprogramming tool and adaptable software tool under the problem areas to which they apply, then evaluating the implications of the groupings. The results are presented in Appendix D and confirm the conclusions given above.

2.7 Postulation of New Microprogramming Tools

The purpose of Task 7 was to postulate a set of microprogram verification and validation tools that would provide significant improvement over existing microprogramming tools and that would aid in resolving the inadequately addressed microprogramming problem areas identified in Task 6. The approach taken was as follows:

- Use the results of Task 5 to identify software tool capabilities that could be effectively incorporated into new microprogramming tools
- Identify capabilities needed for microprogram verification and validation but not found in existing microprogramming or software tools

- o Postulate a set of microprogramming tools that would provide both types of capabilities

Task 5 had shown that tools incorporating the capabilities of AMPIC, BAP, CLARKE, CWRU, DISSECT, EFFIGY, and PET would contribute significantly to microprogram verification and that tools similar to AIDS, CADSAT, DAVE, FACES, and STRUCT would offer additional aid. As Table 1 indicates, these tools provide the following capabilities:

- o Abort diagnosis
- o Breakpoint control
- o Formal verification
- o Simulation/symbolic execution
- o Requirements analysis aid
- o Cross-reference generation
- o Flowchart generation
- o Execution monitoring
- o Test case generation
- o Program auditing

Examination of Table 2 showed that the first four of these capabilities are already offered by existing microprogramming tools. Since no tool offers symbolic execution as its primary function, however, it was decided to retain this capability as one to be implemented in the proposed tools. The fifth item in the list, requirements analysis aid, is a feature of CADSAT, an existing tool that can be applied to either software or firmware development. The remaining five capabilities were determined to be candidates for implementation in the proposed tools. Additional capabilities identified as necessary for addressing the unresolved problem areas were as follows:

- o Analysis of timing and performance
- o Analysis of interruptibility
- o Configuration control aid

The resulting nine capabilities served as the basis for the tool proposal activity. A basic decision made during this activity was to keep the proposed tools independent of one another so that any subset of the tools could be selected for implementation without concern that one tool required output from another. The proposed tools were as follows:

- o Conflict Analyzer
- o Control Flow Analyzer
- o Data Flow Analyzer
- o Execution Monitor
- o Global Cross-Reference Generator
- o Microcode Comparison Program
- o Performance Evaluator
- o Programming Practices Monitor
- o Symbolic Executor

- o Test Case Generator
- o Timing Analyzer

An abstract was developed for each tool, identifying its purpose and describing its proposed capabilities. These abstracts follow.

2.7.1 Conflict Analyzer

The proposed Conflict Analyzer is a static analysis tool designed to identify possible conflicts in the use of computer resources as a result of concurrent operations or interrupts in microprogram execution. Its use may reveal the following types of errors:

- o Use of the same register or memory location by concurrent operations, which may result in loss of intermediate or final results

- o Use of the same register or memory location by routines that may interfere with each other because of hardware- or software-originated interrupts

The Conflict Analyzer prints a report identifying each computer resource that may be misused in this way and the operations or routines that are in contention for the resource. It thereby pinpoints possible resource conflict errors that are difficult to detect and duplicate during microprogram execution.

2.7.2 Control Flow Analyzer

The proposed Control Flow Analyzer is a static analysis tool designed to provide information on the flow of control within a microprogram. It processes the microprogram source code as input and generates two types of output:

- o A flowchart of the microprogram

- o Responses to user questions concerning the flow of control within the microprogram

User questions may include the following:

- o What microinstructions might immediately follow a given microinstruction during microprogram execution?

- o What microinstructions might immediately precede a given microinstruction during microprogram execution?

- o What are all possible paths from one microinstruction to another?

The output of the Control Flow Analyzer is designed to provide insight into the logical structure of the microcode. The flowcharting capability provides a visual representation of the microprogram, aiding in both verification and documentation.

2.7.3 Data Flow Analyzer

The proposed Data Flow Analyzer is a static analysis tool designed to aid in detecting data flow anomalies and in identifying data relationships in a microprogram. By processing the microprogram source code, the Data Flow Analyzer is able to identify the following:

- o Any variable that is used before it is set
- o Any variable that is set but never used
- o Any variable that is reset before the previous value is used
- o All variables affected by a selected assignment to a variable
- o All variables that affect a selected assignment to a variable

The output of the Data Flow Analyzer may indicate logical errors in the microprogram, clerical errors, transpositions of variables or statements, and programming constructs that are subject to error.

2.7.4 Execution Monitor

The proposed Execution Monitor aids in determining the execution characteristics of a microprogram. It operates in two phases. Phase 1 analyzes the control structure of the source program and generates a new source program equivalent to the original except for the addition of code that automatically monitors the execution history of each code segment. This modified version may then be executed in place of the original, generating the desired execution statistics. Phase 2 accepts the execution statistics as input, associates them with the original source code, and presents them in a series of reports. The statistics may also be stored into a history file consisting of data from previous runs. At any point Phase 2 may be requested to incorporate statistics from previous runs into its reports, either for comparison or to provide cumulative results for a series of executions.

2.7.5 Global Cross-Reference Generator

The proposed Global Cross-Reference Generator is a static analysis tool that provides a summary of data usage in a microprogram. The output of the tool lists all data items in the microprogram and provides a description of each item and its use. The description of each item consists of:

- The class of the item: label, data register, status register, variable, routine name, memory, etc.
- If the item is a variable as used in a high-level language, its dimensions, data type, and storage overlay characteristics
- Identification of the routine name and the number of each statement that defines or alters the value of the data item
- Identification of the routine name and the number of each statement that references the data item

Output from the Global Cross-Reference Generator can be used to determine adherence to naming conventions, aid in the debugging process, aid in microcode modification, and provide portions of the microprogram documentation.

2.7.6 Microcode Comparison Program

The proposed Microcode Comparison Program is a tool designed to compare the source code of two versions of a microprogram and to print a report that identifies the differences between them. The two source code versions being compared are designated the "old" version and the "new" version. The Microcode Comparison Program reads the two versions, performs a line-by-line comparison, and identifies the following information:

- Each line of code that has been deleted from the old version to form the new version
- Each line of code that has been added to the old version to create the new version

User-selectable options permit the program to compare selected portions of the two versions and to ignore specified types of source statements, such as blank comment cards. Output from the program can be used to aid in configuration control and in the debugging process.

2.7.7 Performance Evaluator

The proposed Performance Evaluator is used to identify inefficient microcode for possible optimization. The tool is based on a probabilistic model of microprogram behavior developed at the University of Alberta by J. Tartar and S. Dasgupta.*

*Dasgupta, S., and J. Tartar, "The Identification of Maximal Parallelism in Straight-Line Microprograms," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 986-992.

The Performance Evaluator measures three fundamental parameters of microprogram performance:

- o The mean microinstruction reference time (equivalent to the average microinstruction execution time): a function of the control memory clock cycle, the number of accesses to slow memory, the number of branches executed, and whether fetches are serial or overlapped

- o The degree of microprogram parallelism: a measure of the number of microoperations per microinstruction compared to the maximum possible number

- o The branch loss factor: a measure of the effect of branch microinstructions on microprogram performance degradation

The measured parameters are evaluated in a probabilistic manner, allowing expected values of the parameters for typical program execution to be obtained. Execution probabilities are assigned to sequences of microinstructions based on empirical data from microcode simulation runs of typical machine-language programs and on manual analysis. The probabilities are factored into the calculations for reference times, parallelism, and branch loss factors for each of the microinstruction sequences. The results are summed and expected values are obtained for the mean microinstruction reference time, the degree of microprogram parallelism, and the branch loss factor. Results for individual sequences of microinstructions may be examined to detect inefficient sections of code.

2.7.8 Programming Practices Monitor

The proposed Programming Practices Monitor is a static analysis tool designed to detect violations of established programming practices in microprograms. Examples of the types of programming practices this tool may monitor are as follows:

- o Program identification and labeling standards
- o Parameter passing and routine interfacing conventions
- o Variable, register, and label usage conventions
- o Branching standards
- o Conventions for handling instructions and data

The Programming Practices Monitor performs semantic checks of the microprogram beyond those provided by a standard microcompiler or microassembler. It will detect errors, omissions, and questionable code and can improve the quality and uniformity of the code produced.

2.7.9 Symbolic Executor

The proposed Symbolic Executor is a tool that produces symbolic expressions rather than numerical values during microprogram execution. These symbolic expressions describe the transformations performed on specified variables as a result of microprogram execution.

Inputs to the Symbolic Executor are:

- A path specification, identifying the microprogram path to be executed
- Values used to initialize microcode input variables to numeric or symbolic values
- Output requests that identify selected variables and the locations at which their symbolic values are to be displayed

Outputs consist of:

- A path description, indicating the order of the statements executed on the specified path
- A path condition, specifying the restrictions on input variables required for execution of the path
- Symbolic expressions for all requested output variables

The Symbolic Executor is interactive for ease of use. Breakpoints may be inserted, allowing the user to guide the execution based on intermediate results. Comparison of the symbolic results with the expected outcome of each path can aid in microprogram verification.

2.7.10 Test Case Generator

The proposed Test Case Generator is a tool that symbolically executes a microprogram, then uses the results to identify specific input values that will cause selected microprogram paths to be executed. The symbolic execution phase operates as described in Section 2.7.9 above. The test case generation phase uses the path conditions output from the symbolic execution phase. These conditions, in the form of a system of inequalities, are simplified and, if possible, solved. For some cases, the solution process may require user intervention and guidance. The tool is interactive to permit this user participation. The Test Case Generator aids in the verification process by helping to identify input data that can be used to test selected microprogram paths.

2.7.11 Timing Analyzer

The proposed Timing Analyzer is a static analysis tool designed to determine the execution time of selected portions of a microprogram. It accepts as input the source code to be analyzed, a table of timing formulas associated with the microinstructions, and a set of user-prepared path specifications. The types of timing information that may be obtained include the following:

- The minimum time required to execute a selected path
- The maximum time required to execute a selected path
- A representative execution time for a selected path
- A specific execution time for a selected path, based on additional user inputs that permit the exact execution time of variably timed microinstructions to be calculated

The Timing Analyzer may be used to compare alternative implementations of a routine, to identify routines or paths that are unacceptably time consuming, and to display overall timing characteristics of a microprogram.

2.8 Assessment of the Impact of the Proposed Microprogramming Tools on the Microprogramming Problem Areas

Task 8 evaluated the applicability of the microprogramming tools proposed in Task 7 to the microprogramming problem areas identified in Task 3. The results of this evaluation are presented in Table 6. The ratings for the existing micoprogramming tools, as determined in Task 4, are also included so that an assessment of all microprogram verification and validation tools, both existing and proposed, might be presented in one place. The meaning of the ratings in the table is as described in Section 2.4.

As the last row of the table indicates, this set of microprogramming tools provides good coverage of all of the microprogramming problem areas having to do with the microcode itself. The only areas not covered by tools of high applicability are the first and last, which are concerned with errors in specifications and in documentation. No microprogramming tool was proposed for the first problem area because CADSAT, an existing software tool, is applicable. The area of documentation errors, although addressed by several of the proposed microprogramming tools, remains a problem to be handled manually.

2.9 Development of a Tool Selection Metric

The purpose of Task 9 was as follows:

- To identify a set of attributes on which to base the selection of an optimum set of microprogramming tools
- To formulate an algorithm for assigning numerical values to these attributes

The initial approach to this task was to review available literature on software evaluation techniques. The literature search revealed a number

Table 6. Applicability of Existing and Proposed Microprogramming Tools to Microprogramming Problem Areas

Tool	Problem Area											
	1	2	3	4	5	6	7	8	9	10	11	12
BFTSC ICS			Low	Low	Low	Med	Low	Med	Med		Med	
Conflict Analyzer		Med	Low					Med	High			
Control Flow Analyzer		Med	Med			Med				Med		Med
Data Flow Analyzer			Med			Low				Low		Med
Execution Monitor				Low		Med				Med		
Global Cross-Reference Generator		Low	Low	Med	Low	Low					Low	Med
Interactive Debugger			Low	Low	Low	Low	Low	Med	Med		High	
MCS	Low		High	High	High	High	High		Low	High	Low	Low
Microcode Comparison Program			Low									Low
Performance Evaluator		Med						Low				
PRIM Debugger			Low	Low	Low	Low	Low	Med	Med		High	
Programming Practices Monitor		High	Low		Low				Med		Med	
STRUM	Low		High	High	High	High	High		Low	High	Low	Low
Symbolic Executor			High	High	High	High				High		Med
Test Case Generator			High	High	High	High				High		Med
Timing Analyzer			Low					High				Med
Highest rating	Low	High	High	High	High	High	High	High	High	High	High	Med

of comprehensive reports* setting forth attributes that characterize software quality and describing metrics that could be applied to these attributes. Review of these papers, however, led to the conclusion that they did not apply to the Reliable Microprogramming Study. There were three primary reasons for this conclusion:

- o Investigations into software quality focus on programming methods to a far greater extent than is appropriate for tool evaluations. For example, in evaluating the quality of a program, one investigates its consistency in following naming conventions, its freedom from unreachable code, the number of GOTO statements it contains, etc. Although tool quality is crucial, these very specific criteria are not useful in selecting an optimum set of tools.

- o Issues related to practicality, which are crucial to the evaluation of tools, are not addressed in software quality studies. It is not enough to know that a tool is a reliable, correct program; the tool must also be cost-effective to acquire and to use. This requirement takes the evaluation of tools beyond the scope of software quality studies.

- o To evaluate the quality of a program using published metrics, the program must already exist. The approach of the Reliable Microprogramming Study was to identify existing tools, postulate additional ones, and apply a set of attributes to both categories. Many attributes of software quality, such as accuracy and correctness, cannot be applied to tools that have yet to be developed.

Given these concerns, the problem that presented itself was to identify a set of attributes that would permit consideration of both quality and practicality and that could be applied to conceptual as well as existing tools. The following set of attributes was selected:

- o Range of application
- o Effectiveness
- o Evidence of reliability
- o Usability
- o Initial cost
- o Cost of application

*Craig, G. R., W. L. Hetrick, and M. Lipow, Software Reliabiality Study, RADC-TR-74-250, Oct 1974.

McCall, J. A., P. K. Richards, and G. F. Walters, Factors in Software Quality: Concept and Definitions of Software Quality, RADC-TR-77-369, Nov 1977.

Section 2.9.1 defines each attribute and identifies the factors to be considered in applying it to microprogramming tools. Section 2.9.2 discusses the algorithm that was formulated for assigning numerical values to the attributes.

2.9.1 Tool Attributes

2.9.1.1 Range of Application: Range of application is defined as the extent to which the tool can be applied to all microprogram development. Determining factors are:

- o The tool's language independence (i.e., its ability to handle microprograms written in different languages)
- o The nature of the limitations it imposes on the microprograms it is to handle (e.g., size, level of complexity)
- o Its portability (i.e., its ability to execute on more than one computer)

2.9.1.2 Effectiveness: Tool effectiveness is defined as the degree to which the tool can contribute to the verification and validation of a microprogram. Determining factors are:

- o The number of microprogramming problem areas the tool addresses (as defined in Table 3)
- o The usefulness of the results (e.g., the amount of information given, the specificness of the information, the potential for using the tool to improve the microprogram)

2.9.1.3 Evidence of Reliability: Evidence of reliability is defined as the extent to which the tool is known to perform its intended function consistently and with the required precision.

For tools that are to be used as-is or modified only slightly, the determining factor for this attribute is the availabity of knowledge that the tool has undergone comprehensive testing and has a history of producing consistent, accurate results. Tools yet to be developed must be considered to have unknown reliability.

2.9.1.4 Usability: Tool usability is defined as the ease with which it is possible to use the tool. Determining factors are:

- o The ease of formulating tool inputs (e.g., amount of planning, preparation, and expertise required)
- o The ease of applying the tool (e.g., amount of interaction and analysis required during execution)

- o The ease of interpreting the results (e.g., clarity of results, amount of analysis and expertise required)

2.9.1.5 Initial Cost: A tool's initial cost is defined as the cost involved in acquiring, installing, modifying, or developing the tool. The factors used to evaluate this attribute depend upon the tool's status.

If the tool is to be used as-is, its initial cost will depend upon the cost of acquiring and installing the tool and the cost of special equipment and support software needed to support it. If the tool is to be modified from an existing tool, its initial cost will be determined by the cost of acquiring the unmodified tool, the cost of the modification, and the cost of special equipment and support software. If the tool is to be newly developed, its initial cost will depend upon the cost of developing the tool and the cost of special equipment and support software.

2.9.1.6 Cost of Application: The cost of tool application is defined as the cost of using the tool to perform its intended function. Factors that affect this cost are as follows:

- o The computer cost for an average application of the tool
- o The cost to use special equipment
- o The manpower cost associated with using the tool
- o The number of times the tool will be applied to each microprogram

2.9.2 Evaluation Metrics

The second part of Task 9 consisted of formulating an algorithm for assigning numerical values to the attributes described above. The evaluation form in Appendix E presents the results of this effort. Each evaluation factor is to be assigned a numeric value between 1 and 5, with the low values corresponding to the best rating.

Three of the attributes--range of application, effectiveness, and usability--are evaluated by computing the weighted average of the metrics assigned to their contributing factors. We recommend that each factor be weighted equally, but the averaging formula allows for a non-equal weighting.

Both of the cost attributes--initial cost and cost of application--are evaluated by estimating actual costs, then assigning a metric based on the results. The method of estimating initial cost depends upon whether the tool is to be used as-is, modified from an existing tool, or newly developed. The method of evaluating the cost of application takes into account both the cost of a single application and the number of times the tool will be applied to any given microprogram. The product of these two numbers provides an estimate of the total cost of applying the tool to a given microprogram.

The final step in the evaluation of each tool is to compute the weighted average of the metrics assigned to the six attributes. Once again, we recommend an equal weighting of these metrics, but the averaging formula allows for the assignment of any desired weighting factors. It may be desirable to use different weighting factors at different times. By maintaining a record of the metric assigned to each attribute, it will be possible to recompute the average whenever a different weighting is desired.

2.10 Application of the Tool Selection Metric

Task 10 consisted of the following activities:

- o Applying the tool selection metric developed in Task 9 to the existing and proposed microprogramming tools
- o Using the results of this evaluation and factors specific to the RADC microprogramming environment to identify a recommended set of microprogramming tools for RADC

Table 7 presents the results of the first activity. The designations 1a, 1b, etc., across the top correspond to the identifiers given in Appendix E. Where weighted averages have been called for, all factors have been given equal weighting. All of the tools except MCS have been given a rating of "5" under Attribute 3, Evidence of Reliability, because all except MCS would require either new development or extensive modification. The overall score for each tool is given at the far right of the table. Averages have been expressed to two decimal places in order to obtain non-duplicate scores.

Ordering the tools by their scores, with the best-rated placed first, results in the following ranking:

o	Microcode Comparison Program	(2.22)
o	Global Cross-Reference Generator	(2.50)
o	Programming Practices Monitor	(2.72)
o	Control Flow Analyzer	(3.11)
o	Timing Analyzer	(3.17)
o	Conflict Analyzer	(3.19)
o	Data Flow Analyzer	(3.25)
o	BFTSC ICS	(3.36)
o	PRIM Debugger	(3.39)
o	Execution Monitor	(3.50)
o	MCS	(3.58)
o	Performance Evaluator	(3.61)
o	Symbolic Executor	(3.92)
o	Interactive Debugger	(3.94)
o	Test Case Generator	(4.08)
o	STRUM	(4.19)

Table 7. Evaluation of the Existing and Proposed Microprogramming Tools

Tool	Attribute Score														Tool Score
	1a	1b	1c	1	2a	2b	2	3	4a	4b	4c	4	5	6	
BFTSC ICS	1	1	2	1.33	2	3	2.5	5	3	1	3	2.33	4	5	3.36
Conflict Analyzer	5	2	2	3	4	3	3.5	5	1	1	3	1.67	3	3	3.19
Control Flow Analyzer	3	2	2	2.33	3	3	3	5	2	1	1	1.33	4	3	3.11
Data Flow Analyzer	3	2	2	2.33	3	4	3.5	5	2	1	2	1.67	4	3	3.25
Global Cross-Reference Generator	5	2	2	3	3	3	3	5	1	1	1	1	1	2	2.50
Execution Monitor	5	2	2	3	4	4	4	5	1	1	1	1	3	5	3.50
Interactive Debugger	1	1	5	2.33	2	4	3	5	3	4	3	3.33	5	5	3.94
MCS	1	3	3	2.33	2	1	1.5	3	5	4	5	4.67	5	5	3.58
Microcode Comparison Program	1	1	2	1.33	4	4	4	5	1	1	1	1	1	1	2.22
Performance Evaluator	1	2	2	1.67	4	4	4	5	4	1	4	3	3	5	3.61
PRIM Debugger	1	1	5	2.33	2	2	2	5	3	3	3	3	4	4	3.39
Programming Practices Monitor	5	2	2	3	3	3	3	5	1	1	2	1.33	2	2	2.72
STRUM	5	3	3	3.67	2	1	1.5	5	5	5	5	5	5	5	4.19
Symbolic Executor	5	3	3	3.67	3	2	2.5	5	4	2	4	3.33	4	5	3.92
Test Case Generator	5	3	3	3.67	3	2	2.5	5	4	3	3	3.33	5	5	4.08
Timing Analyzer	5	2	2	3	4	4	4	5	3	1	2	2	3	2	3.17

LEGEND FOR ATTRIBUTES

1. Range of application
   1a. Language independence
   1b. Nature of limitations
   1c. Portability
2. Effectiveness
   2a. Problem areas covered
   2b. Usefulness of results
3. Evidence of reliability
4. Usability
   4a. Ease of preparation
   4b. Ease of application
   4c. Ease of interpretation
5. Initial Cost
6. Cost of application

LEGEND FOR SCORES

See Appendix E.

A significant trend revealed by this ranking was that the tools that are less expensive to develop and use, and that require less sophistication on the part of the user, received the better rankings. The better-ranked tools also tend to address some of the more common microprogramming errors, whereas tools at the lower end of the ranking address errors that are more subtle.

The second activity of Task 10 called for the identification of a recommended set of microprogramming tools for RADC. It at first seemed natural to select these tools by choosing the smallest possible group of tools that together covered all of the microprogramming problem areas, and to favor the higher rated of two tools when both addressed the same problem area. Attempts to use this approach, however, showed that two or more tools might have a high degree of applicability to the same problem area without supplying any redundant information and, in fact, that the types of information provided were almost mutually exclusive. As a result, it was not possible to eliminate any tools from the list by using this approach.

Three factors specific to the RADC microprogramming environment provided the final criteria for tool selection. The first of these factors was that RADC is acquiring the Q-PRIM system, an adaptation of PRIM for the QM-1, the computer for which RADC's microprogram development is performed. Since the general capabilities of the BFTSC ICS, the Interactive Debugger, and the PRIM Debugger will all be provided by Q-PRIM, it was determined that these three tools should not be included in the recommended set. The second factor was that Logicon had been directed by RADC to consider proof-of-correctness techniques outside the scope of the project. This directive resulted in the elimination of STRUM and MCS. Finally, Logicon had been directed to deemphasize tools that would focus on the nanocode level of the QM-1. This directive resulted in elimination of the Conflict Analyzer and the Performance Evaluator.

The remaining nine tools were determined to be a nonredundant, cost-effective set of tools for microprogram verification and validation in RADC's environment, and were selected as the recommended set. These tools, listed in order with the best-ranked first, are as follows:

- Microcode Comparison Program
- Global Cross-Reference Generator
- Programming Practices Monitor
- Control Flow Analyzer
- Timing Analyzer
- Data Flow Analyzer
- Execution Monitor
- Symbolic Executor
- Test Case Generator

Logicon recommends that, within funding and other practical constraints, these tools be developed in the order given above.

3. THE TOOL SPECIFICATION ACTIVITY

The second part of the Reliable Microprogramming Contract was devoted to the development of functional specifications and usage scenarios for the recommended microprogramming tools. Section 3.1 describes the development of the functional specifications; Section 3.2 presents the usage scenarios.

3.1 Development of Functional Specifications

The first task of the tool specification activity was the development of functional specifications for the recommended tools. This task required that the general tool concepts developed during the study period be applied specifically to the RADC microprogramming environment.

Microprogram development at RADC is performed for the QM-1 computer, using two languages:

- MULTI, which resembles assembly language
- SMITE, a higher-order language used for emulation

Since all of the tools except the Microcode Comparison Program analyze and interpret microprogram source code, a decision had to be made for each tool as to whether it should be directed to microprograms written in MULTI, those written in SMITE, or both. Table 8 identifies the decision made for each tool.

As the table indicates, five of the tools will be directed to microprograms written in either language. The Microcode Comparison Program will be language independent. The Control Flow Analyzer and the Data Flow Analyzer will each have a MULTI preprocessor and a SMITE preprocessor providing language-independent information to their main processing functions. Two separate Programming Practice Monitors and two separate Global Cross-Reference Generators are recommended. Since MULTI does not offer the capability of explicitly defining subroutines, the distinction between global and local variables does not exist, and the MULTI Cross-Reference Generator will not have the word "global" in its name.

The remaining four tools will be directed to a single language. The Timing Analyzer was restricted to MULTI because its processing depends upon its knowledge of the specific microinstructions that constitute each microprogram path. This knowledge is lacking with a higher-order language such as SMITE. The Execution Monitor, the Symbolic Executor, and the Test Case Generator were restricted to SMITE. This decision was made for the Execution Monitor because instrumentation of MULTI code at every branch would result in unacceptable overhead at execution time. The decision was made for the Symbolic Executor and the Test Case Generator because many of the functions performed by MULTI are not well suited to expression in the symbolic notation on which these tools are based.

Table 8. Languages Processed by Each Tool

Tool	Language Processed MULTI	Language Processed SMITE	Comments
Microcode Comparison Program	X	X	A single, language-independent tool
Global Cross-Reference Generator	X	X	Two separate tools
Programming Practices Monitor	X	X	Two separate tools
Control Flow Analyzer	X	X	Two preprocessors to a single tool
Timing Analyzer	X		Not suited to a higher-order language
Data Flow Analyzer	X	X	Two preprocessors to a single tool
Execution Monitor		X	Excessive overhead if used on MULTI
Symbolic Executor		X	Some MULTI functions unsuited to symbolic expression
Test Case Generator		X	Some MULTI functions unsuited to symbolic expression

Discussions with RADC personnel disclosed that specification of Programming Practice Monitors for MULTI and SMITE would be premature at this time. As an alternative, a list of programming practices that might be monitored by these tools was developed for future reference. The results of this activity are presented in Appendix F.

Two documents were prepared for each of the other tools:

- o A Functional Description, intended to reflect the initial definition of the tool and to provide users with a clear statement of its operational capabilities
- o A System Specification, intended to fully define the requirements of the tool and to provide development personnel with the information required for its design

Each document was developed in accordance with DoD Automated Data System Documentation Standards Manual 4120.17-M. They appear in the following appendixes:

G,H - Microcode Comparison Program
I,J - MULTI Cross-Reference Generator
K,L - SMITE Global Cross-Reference Generator
M,N - Control Flow Analyzer
O,P - Timing Analyzer
Q,R - Data Flow Analyzer
S,T - Execution Monitor
U,V - Symbolic Executor
W,X - Test Case Generator

Paragraphs common to all of the Functional Descriptions are given in Appendix G only; paragraphs common to all of the System Specifications are given in Appendix H only. The remaining appendixes reference these paragraphs as required.

Each of the tools was specified as an independent system so that any subset of the tools could be selected for development. It should be noted, however, that a number of the tools will have functions in common and that the development times specified in the Functional Descriptions could be significantly reduced if tools with functions in common were developed together. Examples of shared functions are as follows:

- o The Control Flow Analyzer, Timing Analyzer, Data Flow Analyzer, Symbolic Executor, and Test Case Generator will all have preprocessors that analyze source code and generate an internal model of the microprogram being processed.
- o The Test Case Generator will contain a Symbolic Executor function similar to the Symbolic Executor system itself.

- o The Cross-Reference Generators for SMITE and MULTI will have functions in common, as will the two Programming Practice Monitors.

Identification of these common functions and selection of a development plan that takes advantage of them could result in a considerable reduction in the specified development times.

3.2 Usage Scenarios

The second task of the tool specification activity was to describe the manner in which the recommended tools might be used. The results of this task follow.

3.2.1 Microcode Comparison Program

The purpose of the Microcode Comparison Program is to identify all of the differences between two versions of a microprogram or to establish that two versions are in fact identical. The primary applications of this tool will be for configuration control and, to a lesser degree, for microprogram debugging. Configuration control applications will include providing development and testing personnel with an accurate accounting of changes as they are made, demonstrating that the certified version is identical to the version that was verified, and determining whether an operational version is identical to the certified version. Debugging applications will include pinpointing the source of an error by identifying the differences between a version of the microprogram that is experiencing problems and a version in which these problems do not exist.

3.2.2 MULTI Cross-Reference Generator

The MULTI Cross-Reference Generator will provide an alphabetical listing of all labels, registers, and symbolic names used in a MULTI microprogram. This listing will provide descriptive information about each entry and will identify the statement in which the entry is declared, the statements in which its value is changed, and the statements in which its value is used. The primary applications for this tool will be for debugging, documentation, and microprogram maintenance. The user may refer to the cross-references to:

- o Determine the characteristics of a particular entry, the statements that establish and alter its value, and the statements that use its value
- o Trace the source of an error detected during testing
- o Trace the effects of proposed changes to the microprogram
- o Identify relationships that exist among the entries

- Avoid introducing duplicate symbolic names when making modifications
- Identify unreferenced entries
- Verify adherence to naming conventions

Because of their usefulness, the cross-references will serve as a permanent part of the microprogram documentation.

3.2.3 SMITE Global Cross-Reference Generator

The SMITE Global Cross-Reference Generator will provide global cross-references for the variables, labels, and processors used in a SMITE program. These cross-references, like those produced by the MULTI Cross-Reference Generator, will provide descriptive information about each entry and will identify the statement in which the entry is declared, the statements in which its value is changed, and the statements in which its value is used. The applications of this tool will be similar to those described for the MULTI Cross-Reference Generator, namely, debugging, documentation, and microprogram maintenance. An additional capability will be to make visible the setting and use of variables that have been equivalenced through use of the DEFINED attribute in SMITE.

3.2.4 MULTI Programming Practices Monitor

The MULTI Programming Practices Monitor will process the source code of a microprogram written in MULTI and identify violations of established programming practices. This tool can be applied to newly written or modified microprograms and will result in the following benefits:

- Disclosure of coding errors and error-prone constructions
- Greater uniformity in coding style among the members of a development team
- Better identification and labeling of the microprogram

Each of these benefits can aid in both debugging and maintenance of the microprogram.

3.2.5 SMITE Programming Practices Monitor

The SMITE Programming Practices Monitor will process the source code of a SMITE program and identify violations of established programming practices. Like the MULTI Programming Practices Monitor, its use will result in error and problem disclosure, greater uniformity in coding style, and better program labeling.

3.2.6 Control Flow Analyzer

The Control Flow Analyzer will perform the following functions:

- o Analyze the flow of control within a microprogram written in MULTI or SMITE
- o Generate a flowchart of the microprogram
- o Generate a structured flowchart of the microprogram
- o Identify any unreachable code
- o Respond to user requests to identify all possible predecessors and successors to given statements and all possible paths between selected pairs of statements

The flowcharts that are generated may be used as microprogram documentation and as an aid in familiarization, analysis, test planning, debugging, and microprogram maintenance. By displaying the logical structure of the microprogram in a way that a code listing cannot, they will aid in identifying logical errors and in determining the effects of proposed changes to the microprogram. The structured flowcharts will give further insight into the logic of the microprogram. Identification of unreachable code will serve to pinpoint logical errors and suggest code improvement.

The response-generating capability of the tool can be used for debugging and maintenance. It will be an interactive capability that will enable the user to investigate the logical structure of the microprogram, trace forward and backward in the microprogram to determine the source and impact of errors detected during testing, and investigate the impact of proposed changes on the microprogram logic.

3.2.7 Timing Analyzer

The Timing Analyzer will use a table of timing formulas to determine the execution time of selected portions of a microprogram written in MULTI. The user will interactively select a set of paths through the microprogram, and the tool will compute the amount of time required for execution of each path.

The Timing Analyzer can be used during microprogram verification to aid in identifying paths that are unacceptably time consuming or that are candidates for optimization. It can also be used in microprogram development and maintenance to compare the execution times of alternative implementations of a given algorithm. Finally, the timing characteristics it generates can become part of the micropr‿gram documentation.

3.2.8 Data Flow Analyzer

The Data Flow Analyzer will perform the following functions:

- Analyze the flow of control and the data usage properties of a microprogram written in MULTI or SMITE
- Identify data flow anomalies, namely, instances of variables being used before they are set, set but never used, or reset before being used
- Respond to user questions concerning the effects of variable settings on one another during microprogram execution

The Data Flow Analyzer can be used for microprogram debugging and maintenance. Applied to a newly developed or modified microprogram, its anomaly detection capability will result in early discovery of logical and clerical errors that might otherwise cause problems in testing. The response-generating capability can be used interactively to identify all possible variable assignments that may have caused a particular error detected during testing and to determine the effects on microprogram variables of possible modifications to the code.

3.2.9 Execution Monitor

The Execution Monitor will perform the following functions:

- Modify a SMITE program in such a way that execution statistics will be generated whenever the program is executed
- Maintain a history file of these statistics and generate detailed and summary reports on them

The execution statistics that are generated will indicate the number of times each segment of code was executed or will identify the segments that were executed at least once.

The Execution Monitor will have a number of applications. It will enable the user to identify untested portions of a program so that test cases can be devised to exercise these portions. It will also identify portions of the program that are heavily used and are therefore candidates for optimization. Finally, the statistics that are generated will indicate execution patterns that may pinpoint unexpected behavior and logical erors that would otherwise be difficult to isolate.

3.2.10 Symbolic Executor

The Symbolic Executor will simulate the execution of a SMITE program, using symbolic, rather than numeric, values for program variables. It will operate interactively to execute a set of paths specified by the user and will produce as output a path description, the conditions that caused each path to be executed, and the symbolic values of selected program variables. Since the inputs and outputs will be in the form of symbolic expressions, the symbolic execution will present the results for an entire class of inputs.

The Symbolic Executor can be used for SMITE program verification. Manual inspection of the results of each execution will enable the user to determine whether the selected paths have performed their intended functions and produced the expected results. In addition, the path conditions, if not too complex, can be used to identify actual data values that will cause the selected paths to be traversed during normal execution of the program.

3.2.11 Test Case Generator

The Test Case Generator will combine the capabilities of a Symbolic Executor with the added capability to identify actual values that will satisfy path conditions. The results will be input values that will cause the execution of selected paths of a SMITE program. The tool will be interactive to permit user participation in the selection of paths during symbolic execution and the isolation of solutions for the path conditions.

The Test Case Generator can be used as an aid in the testing process. The uses for its symbolic execution capabilities will be as described in Section 3.2.10. The test case generation capability can be used to produce specific test cases to exercise paths of particular interest for which selection of suitable input values is difficult to achieve manually.

4. CONCLUSIONS AND RECOMMENDATIONS

The Reliable Microprogramming Study analyzed current software and microprogramming tools, identified problem areas in microcode development, determined the applicability of current tools to the microprogramming problem areas, and proposed additional microprogramming tools. A tool selection metric was developed and was used, together with factors from the RADC microprogramming environment, to identify a recommended set of microprogramming tools for RADC. The requirements for these tools were identified, and functional specifications were prepared.

The following conclusions resulted from the study:

- A wide variety of automated tools are available to aid in the verification and validation of software, while tools for microprogram verification and validation are limited in both number and capability.

- Microprogram development problem areas do not differ substantially from software development problem areas, although the emphasis may differ in the two types of development.

- Existing software tool technology is transferable to a microprogramming environment and constitutes a solid foundation for the development of microprogram verification and validation tools.

- Existing and proposed tools can be quantitatively evaluated and compared by using a carefully defined set of tool attributes.

The RADC directive to focus on the microcode, rather than nanocode, level of QM-1 microprogramming had a significant impact on the study outcome. Most of the issues that distinguish microprogramming from conventional programming arise at the nanocode level. These issues, which result from the parallelism inherent in nanocode, include problems of resource conflicts, concurrency of operations, and timing of control signals. It is at this level that microprogramming differs significantly from conventional programming. QM-1 microcode instructions are similar to machine instructions except that they invoke simpler sequences of control signals. They are considerably easier to design and implement than nanocode and typically specify serial rather than parallel operations.

The RADC environment also influenced the types of tools that were recommended. The similarity of microcode development in MULTI to assembly language programming and the existence of the higher-order emulation language SMITE resulted in the recommendation of microprogramming tools based largely on software tool concepts. The recommended tools are as follows:

- Microcode Comparison Program
- Global Cross-Reference Generators for MULTI and SMITE
- Programming Practice Monitors for MULTI and SMITE
- Control Flow Analyzer
- Timing Analyzer
- Data Flow Analyzer
- Execution Monitor
- Symbolic Executor
- Test Case Generator

Logicon recommends that, within funding and other constraints, these tools be developed in the order shown above. Development of the Programming Practice Monitors, however, should be delayed until sufficient development work has been performed in MULTI and SMITE for programming practices to have been well established. It is recommended that these monitors be modular in structure to facilitate the addition and deletion of specific monitoring capabilities to accommodate changes in the microprogramming environment.

It is also recommended that the tools identified above be developed in a way that takes advantage of similar and shared functions. This approach will minimize the development costs.

REFERENCES

1. Birman, A., "On Proving Correctness of Microprograms," IBM Journal of Research and Development, May 1974, pp. 250-266.

2. Britt, B., et al., PRIM System: Overview, ARPA Order No. 2223, ISI/RR-77-58, Information Sciences Institute, Mar 1977.

3. Brown, J. R., and K. F. Fischer, "A Graph Theoretic Approach to the Verification of Program Structures," Third Int. Conf. on Software Engineering, 10-12 May 1978.

4. Brown, W. S., Altran User's Manual, Vol. 1, Bell Telephone Lab, 1973.

5. Clarke, L. A., A System to Generate Test Data and Symbolically Execute Programs, Univ. of Colorado, Rept. CU-CS-060-75, Feb 1975.

6. Clarke, L. A., "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation," Ph.D. Thesis, Univ. of Colorado, 1976.

7. Clarke, L. A., "A Program Testing System," Proc. of the Annual Conf. of the ACM, Houston, TX, 20-22 Oct 1976.

8. Craig, G. R., W. L. Hetrick, and M. Lipow, Software Reliability Study, RADC-TR-74-250, Oct 1974, 787784/8GI.

9. Dasgupta, S., and J. Tartar, "The Identification of Maximal Parallelism in Straight-Line Microprograms," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 986-992.

10. DeWitt, D. J., M. S. Schlansker, and D. E. Atkins, "A Microprogramming Language for the B-1726," Conf. Record of the Sixth Annual Workshop on Microprogramming, College Park, MD, 24-25 Sep 1973, pp. 21-39.

11. Evans, Moffett, and Merwin, "Design of Assembly Level Language for Horizontal Encoded Microprogrammed Control Unit," Conf. Record of the Seventh Annual Workshop on Microprogramming, Palo Alto, CA, 30 Sept-2 Oct 1974, pp. 217-224.

12. Fosdick, L. D., and L. J. Osterweil, "DAVE-A FORTRAN Program Analysis System," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

13. Fosdick, L. D., and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys, Vol. 8, No. 3, Sep 1976.

14. Gallenson, L., et al., PRIM System: AN/UYK-20 User Guide, User Reference Manual, ARPA Order No. 2223, ISI/TM-77-5, Information Sciences Institute, Oct 1977.

15. Gallenson, L., et al., PRIM System: Tool Builder's Manual, User Reference Manual, ARPA Order No. 2223, ISI/TM-78-7, Information Sciences Institute, Jan 1978.

16. Gasser, M., "An Interactive Debugger for Software and Firmware," Conf. Record of the Sixth Annual Workshop on Microprogramming, College Park, MD, 24-25 Sep 1973, pp. 113-119.

17. Grishman, R., "Criteria for a Debugging Language," Debugging Techniques in Large Systems, Randall Rustin, Ed., pp. 57-75.

18. Hardy, I. T., B. Leong-Hony, and D. W. Fife, Software Tools: A Building Block Approach, NBS Special Publication 500-14, NTIS PB 270-971.

19. Hoffman, R. H., "NASA/Johnson Space Center Approach to Automated Test Data Generation," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

20. Hoffman, R. H., User Information for the Interactive Automated Test Data Generator (ATDG) System, NASA Document JSC-10832, 15 Jan 1976.

21. Hoffman, R. H., "An Interactive Automated Test Data Generator," Proc. of the Annual Conf. of the ACM, Houston, TX, 20-22 Oct 1976.

22. Hollowich, M. E., and M. G. McClimens, Software Design and Verification System, Air Force Avionics Laboratory Tech. Rept. AFAL-TR-76-200, AD A042126.

23. Hookway, R. J., A Program Verification System, Report No. 1171, Jennings Computing Center, Case Western Reserve University, Jan 1976.

24. Hookway, R. J., and G. W. Ernst, "A Program Verification System," Proc. of the Annual Conf. of the ACM, Houston, TX, 20-22 Oct 1976.

25. Howden, W. E., "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, Vol. C-24, May 1975.

26. Howden, W. E., "Automated Program Validation Analysis," Proc. of the Fourth Texas Conf. on Computing Systems, 17-18 Nov 1975.

27. Howden, W. E., "The DISSECT Symbolic Evaluation System," Proc. of the 1976 National Comp. Conf., New York, NY, 6-10 Jun 1976.

28. Howden, W. E., Symbolic Testing - Design Techniques, Costs and Effectiveness, NTIS PB268-517.

29. Howden, W. E., and J. Laub, "Automatic Case Analysis of Programs," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

30. Ikezawa, M., G. Small, R. Kayfes, R. Galvan, V. Ratzlaff, and F. Stelle, Litton/AMPIC User's Manual, Logicon Report CSS-77002, 18 Apr 1977.

31. Itoh, D., and T. Izutani, "FADEBUG-1, A New Tool for Program Debugging," IEEE Symp. on Computer Software Reliability, 1973.

32. Joyner, W. H., et al., Automated Verification of Microprograms, IBM Research Report FC 5941 (#25780), 12 Apr 1976.

33. King, J. C., "A Program Verifier," Proc. IFIP Cong. 71, North-Holland, Amsterdam, 1971.

34. King, J. C., "Proving Programs to be Correct," IEEE Transactions on Computers, C-20, Nov 1971.

35. King, J. C., "A New Approach to Program Testing," Proc. of the Int. Conf. on Reliable Software, Los Angeles, CA, 21-23 Apr 1975.

36. King, J. C., "Symbolic Execution and Program Testing,," Communications of the ACM, Vol. 19, No. 7, Jul 1976.

37. King, J. C., and R. W. Floyd, "An Interpretation Oriented Theorem Prover Over Integers," J. Computer Syst. Sci., 6 Aug 1972.

38. Leeman, G. B., et al., An Automated Proof of Microprogram Correctness, IBM Research Report RC 6587 (#28423), 20 Jun 1977.

39. McCall, J. A., P. K. Richards, and G. F. Walters, Factors in Software Quality: Concept and Definitions of Software Quality, RADC-TR-77-369, Nov 1977, A049014, A-49015, A049055.

40. McCarthy, P., and J. A. Painter, "Correctness of a Compiler for Arithmetic Expressions," Proc. of the Symp. on Appl. Math., American Math Soc., Vol. 19, 1967.

41. Moore, C. R., User Requirements Analyzer Version 2.0 User's Manual for IBM 370/158/OS/TSO Installation, Released by ISTAO, Electronic Systems Division, Air Force Systems Command.

42. Panzl, D. J., "Test Procedures: A New Approach to Software Verification," Second Int. Conf. on Software Engineering, 13-15 Oct 1976, pp. 477-485.

43. Panzl, D. J., "Automatic Software Test Drivers," Computer, Apr 1978.

44. Panzl, D. J., "Automatic Revision of Formal Test Procedures," Third Int. Conf. on Software Engineering, 10-12 May 1978.

45. Patterson, D. A., "Verification of Microprograms," Ph.D. Thesis, Department of Computer Science, UCLA, 1976.

46. Patterson, D. A., "STRUM: Structured Microprogram Development System for Correct Firmware," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 974-985.

47. Ramamoorthy, C. V., and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, Mar 1975.

48. Ramamoorthy, C. V., R. E. Meeker, and J. Turner, "Design and Construction of an Automated Software Evaluation System," IEEE Symp. on Computer Software Reliability.

49. Ramamoorthy, C. V., et al., "A Higher Level Language for Microprogramming," Conf. Record of the Sixth Annual Workshop on Microprogramming, College Park, MD, 24-25 Sep 1973, pp. 139-144.

50. SMITE Installation and Analysis: SMITE Training Manual, TRW Defense and Space Systems Group 30417-6002-RU-00, RADC-TR-77-364, 12 Aug 1977, A049038.

51. Stucki, L. G., "Automatic Generation of Self-Metric Software," IEEE Symp. on Computer Software Reliability, 1973.

52. Tsuchiya, M., and M. Gonzales, "Toward Optimization of Horizontal Microprograms," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 992-999.

53. Wheatley, R., "A Simulation Technique in Microprogram Validation," Proc. of AIAA Computers in Aerospace Conference, Los Angeles, CA, 31 Oct-2 Nov 1977.

54. Wheatley, R., and E. Yoshizawa, Branch Analysis Program (BAP) User's Guide, Contract F04704-77-C-0001, Logicon, 10 Dec 1976.

APPENDIX A
ABSTRACTS OF CURRENT SOFTWARE TOOLS

The abstracts included here were written under the assumption that the reader is familiar with the software verification and validation vocabulary. Terms such as "symbolic execution," "execution tree," "resolution theorem proving," etc., are used without definition.

A tool classification presented by I. T. Hardy et al.* is given below to serve as a top-level description of the tools presented. A tool is placed in a category if one of its prime functions in some way provides the capability represented by that category. The relevant categories are as follows:

1) Abort Diagnosis Tool: Provides a dump of registers and memory whenever an abnormal termination of program execution occurs.

2) Breakpoint Controller: Allows the user to specify points in a program where execution is to pause and wait for the user to interactively observe and/or change the status of program values.

3) Cross-Reference Generator: Provides a listing of variables and routines, indicating where they are referenced.

4) File/Library Manager: Provides for organized and economical storage of programs and/or data, monitoring their use and change.

5) Flowchart Generator: Produces a pictorial representation of program control flow and/or data flow.

6) Program Auditor: Checks programs for consistency with specified standards; analyzes programs for static characteristics.

7) Execution Monitor: Instruments source code to provide for data collection during program execution. Data collected may include frequency counts for branches and segments, ranges of values which variables assume, execution traces, and snapshot dumps.

*Hardy, I. T., B. Leong-Hony, and D. W. Fife, Software Tools: A Building Block Approach, NBS Special Publication 500-14, NTIS PB270-971.

8) Test Case Generator: Generates a test data file or description based on user specification and/or source program characteristics.

9) Simulator/Symbolic Executor: Simulates program execution by analysis with actual or symbolic variables and subroutines. User may select and control the course of action during execution.

10) Formal Verifier: Provides formal proof that a program or program path agrees with corresponding specifications.

11) Requirements Analysis Aid: Enforces completeness and consistency in requirements definition.

The tools abstracted in this appendix fall into the following categories:

o	Abort Diagnosis Tools	AIDS
o	Breakpoint Controllers	AIDS, AMPIC, EFFIGY
o	Cross-Reference Generators	ACES, AMPIC
o	File/Library Managers	SDVS
o	Flowchart Generators	AMPIC
o	Program Auditors	DAVE, FACES, STRUCT
o	Execution Monitors	ACES, AIDS, BAP, PET
o	Test Case Generators	ATDG, CLARKE
o	Simulators/Symbolic Executors	AMPIC, AUT, DISSECT, EFFIGY, FADEBUG, SDVS, TPL
o	Formal Verifiers	CLARKE, CWRU, EFFIGY
o	Requirements Analysis Aids	CADSAT

ACES

The Automated Code Evaluation System (ACES) is a language processor which performs lexical and syntactical analysis of source programs written in the FORTRAN-like language CENTRAN. It automates some portions of the validation process and provides output to facilitate analysis of the program. ACES consists of three phases:

1) Lexical Analysis and Data Base Generation: In this phase, the program is analyzed and represented in a data base in the form of four tables: symbol table, symbol use table, statement type table, and an abbreviated connection matrix. The data base provides complete static information on all variables and statements. The user is permitted to specify that variable monitoring is to take place. By submitting a list of variables and their upper and lower bounds, the user instructs ACES to modify the source by inserting code which monitors each assignment to the specified variables. If at run time a variable is assigned a value outside the provided bounds, a message is generated.

2) Structure Analysis: In this phase, the program structure is modeled as a directed graph, allowing determination of undefined labels, unreferenced labels, unreachable statements, terminating statements, and program loops. ACES also generates for each statement a list of other statements which may follow in an execution.

3) Report Writing: In this phase, several outputs are provided, including cross-reference listings, warning messages, structural characteristics, etc.

ACES was written in FORTRAN for the CDC 6600, but may be used on the IBM 360 or Univac 1108 with minor modifications. ACES processes about 15 statements per second on the 1108. The static analysis portion of ACES is complete and available as a usable product. The dynamic analysis portions are not operational.

Reference:

Ramamoorthy, C. V., R. E. Meeker, and J. Turner, "Design and Construction of an Automated Software Evaluation System," IEEE Symp. on Computer Software Reliability.

AIDS

The AIDS system provides for interactive debugging of FORTRAN and assembly language programs by using a machine simulator to monitor the user's program. The program to be debugged is submitted to a standard compiler (assembler). AIDS then accesses the compiler-produced listing and object program in order to process user debugging commands. The commands describe program events and specify what actions are to take place when the events occur. Five types of events can be described:

o Execution of a particular operation code
o Execution of an instruction at a specified location
o Stores to a specified location

- o Loads from a specified location
- o Calls to a specified location

The location may be specified by statement number, variable name, subscripted variable name, or routine name.

The actions that may be taken when an event occurs include the following:

- o Print a one-line description of the event
- o Print the contents of variables
- o Print a snapshot of the registers
- o Transfer control to another statement
- o Pause for user input

Any of these actions may be made conditional by prefacing it with a logical IF. When the system pauses for user input, several other commands may be entered, including:

- o Execute the next n instructions and then pause.
- o Restore the program state to what it was before the execution of the last n statements.
- o Change from simulation to direct execution at specified locations.

The AIDS system is operational on the CDC 6600.

Reference:

Grishman, R., "Criteria for a Debugging Language," Debugging Techniques in Large Systems, Randall Rustin, Ed., pp. 57-75.

AMPIC

The Automatic Machine Process Inference Confirmer (AMPIC) is a Logicon-developed tool providing many forms of program analysis. It processes IBM 360 FORTRAN and assembly language and Litton 4516 assembly language. It may be used in batch or time-shared modes. AMPIC consists of seven processors, each providing program analysis output.

- o PREP: FORTRAN or assembly language source is read and processed to provide a model of the program to the other processors. FORTRAN programs are checked for uninitialized variables, unused variables, jumps into DO-loops, etc.
- o LIFE: Assembly programs are segmented into straight-line (S) sections and conditional branch (T) elements. A global cross-reference listing may be requested. Execution times for all S and T elements are provided.

- o <u>WFF</u>: The input program structure is analyzed and a flowchart is provided in terms of S and T elements. An option is available to automatically restructure the program to conform to the single-entry, single-exit rule of well-structured programs.
- o <u>TAP</u>: Minimum and maximum execution times are generated for entire subroutines or groups of subroutines.
- o <u>TFLOW</u>: Flowcharts with enclosed text are output to the line-printer or Calcomp plotter.
- o <u>SPFF</u>: Assembly code is decompiled into symbolic equations. S-element translations relate the input and output variables to each other. T-element translations represent the test condition in terms of the input variables to the T-element.
- o <u>PPFF</u>: The path analyzer symbolically executes a set of paths defined by the user. Each path definition may generate anywhere from a single path to all paths in the program, depending on the detail of the definition. For each path, the system generates a symbolic predicate indicating the input conditions under which the path is executed, a sequence of S and T elements describing the path, and a symbolic value for each output variable.

AMPIC is operational on the IBM 360.

<u>Reference</u>:

Ikezawa, M., G. Small, R. Kayfes, R. Galvan, V. Ratzlaff, and F. Stelle, <u>Litton/AMPIC User's Manual</u>, Logicon Report CSS-77002, 18 Apr 1977.

<u>ATDG</u>

The Automated Test Data Generator (ATDG) is used to support testing of FORTRAN programs. It consists of two functions: 1) static error analysis, used as an aid to initial debugging and desk-checking, and 2) test data generation, used as an aid in designing test cases. ATDG is being developed as an on-going TRW project for the NASA/Johnson Space Center.

Static error analysis is performed in a path-oriented manner similar to that of the DAVE system. It identifies unused assignments to variables, uninitialized variables, unreachable code, and infinite loops.

The primary function, test data generation, identifies predicate conditions necessary to follow desired execution paths. The objective of this function is to generate the fewest number of paths required to collectively exercise all predicate outcomes and to verify the executability of these paths.

Two outputs are provided by the test data generator. The first is a description of the path it has selected in an effort to find the path which executes as many conditional transfers as possible. This output takes the form

```
          LINE:CONDITION          BRANCH EXPRESSION
                .                         .
                .                         .
                .                         .
```

where LINE refers to a line number in the compiled listing and CONDITION specifies the truth value the BRANCH EXPRESSION in that line must assume for the selected path to be executed. The second output describes the effectiveness of the path in terms of what percentage of transfers and statements were executed.

ATDG is interactive; all user inputs are responses to ATDG queries. Optional capabilities are presented in three levels of detail to tailor the system to users of differing experience. For a particularly complex problem, ATDG may ask the user for help. At points where the test data generator would make an arbitrary decision, the user is allowed to provide information relevant to that decision.

ATDG is most useful for processing a single subroutine, but it is aware of external references, and future development will enable it to process programs with multiple routines. It is operational on the Univac 1110 and 1108 under the EXEC 8 operating system.

References:

Hoffman, R. H., "NASA/Johnson Space Center Approach to Automated Test Data Generation," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

Hoffman, R. H., User Information for the Interactive Automated Test Data Generator (ATDG) System, NASA Document JSC-10832, 15 Jan 1976.

Hoffman, R. H., "An Interactive Automated Test Data Generator," Proc. of the Annual Conf. of the ACM, Houston, TX, 20-22 Oct 1976.

AUT

The Automated Unit Test (AUT) system was developed at IBM. When supplemented with test procedures coded in the Module Interface Language-Specific (MIL-S), it provides for the testing of object modules and is independent of the source language. The object module may consist of one or more routines, up to a whole program. Execution of a MIL-S test procedure yields a report of errors in the object module's outputs.

A test procedure coded in MIL-S consists of at least one test case. Each test case consists of: 1) specification of actual values for the arguments to the routines, 2) actual values corresponding to what the output values should be for the given arguments, and, optionally, 3) instructions to intercept calls to other routines. If such calls are to be accepted, the user must supply input and output values for AUT to verify during execution.

Since the user has supplied all correct interface values, AUT can verify that these values are assigned in an execution of the object module. If a discrepancy appears, AUT prints an error message, corrects the error using the supplied value, and continues.

AUT allows program testing to take place independent of the source language used. Since the test procedures are coded in MIL-S, they are well-defined and easy to reproduce. However, the MIL-S language does resemble assembly languages, and test procedures tend to be lengthy. Also, there is no facility for modeling I/O devices and files.

Reference:

Panzl, D. J., "Automatic Software Test Drivers," Computer, Apr 1978.

BAP

The Branch Analysis Program (BAP) was developed by Logicon to support the verification and validation of the Minuteman Operational Targeting Program. BAP is an execution monitor for FORTRAN programs, and is operational on IBM, CDC, and Univac machines. It consists of a preprocessor written in SNOBOL and assembly language and a postprocessor written in FORTRAN and assembly language.

The preprocessor accepts a sequential or partitioned data set containing the input source code. Control commands allow the user to specify input routines that are to be included or excluded and temporary updates that are to be made to the routines before processing begins. The source code is then instrumented with appropriate code to monitor all branch and segment executions and is written to a sequential file. The added code consists of calls to subroutines which increment execution counters. These subroutines are appended to the output file.

The instrumented code is compiled and executed with the user's support software. The program may be executed more than once without loss of data. Execution statistics, automatically written by the modified program, are analyzed by the postprocessor.

The postprocessor provides reports on the execution statistics and permits the statistics to be stored in a history file. Data saved in the history file may be incorporated into the output of future runs, thereby allowing the user to accumulate his execution statistics over a period of time. Three forms of output may be requested for a given set of execution statistics:

- o <u>Frequency</u>: The execution counts for each branch and segment are printed for all monitored routines.
- o <u>Summary</u>: Execution percentages are printed for all monitored routines.
- o <u>Listing</u>: Monitored routines are listed, with an execution count appearing beside each statement and branch.

The BAP system is restricted to at most 300 subroutines consisting of at most 7200 code segments. Up to 50 routines may be included or excluded, and at most 50 previously created statistic members may be requested from the history file.

<u>Reference</u>:

Wheatley, R., and E. Yoshizawa, <u>Branch Analysis Program (BAP) User's Guide</u>, Contract F04704-77-C-001, Logicon, 10 Dec 1976.

CADSAT

The Computer Aided Design and Specification Analysis Tool (CADSAT) is a tool used in creating, analyzing, and maintaining user requirements for a particular system. It addresses two objectives of requirements analysis. The first objective is to improve the quality of the system design by automatically enforcing consistency and clarity as the design proceeds. Outputs provide the user with an easy method of detecting incompleteness and inconsistencies in the design. The second objective is to minimize the design cost and effort required to generate reports. Several report options are available to aid the analyst and the project management, and to generate final specifications reports.

The user requirements are best entered into a user requirements data base by using the TSO facility of the IBM 360/370 series. Several of CADSAT's functions which manipulate and anayze this data base are as follows:

- o As data are entered, semantic and syntactic checks are performed to guarantee the validity of the command and data. Appropriate diagnostics are provided in case of error.

- o Any data in the data base may be rearranged, sorted, or formatted to comply with external documentation requirements.

- o Contextual checks of all input data are performed to guarantee consistency of the structure and attributes of data base entries.

- o Extensive analysis of relationships among data base items is performed. Such analysis would normally be prohibitive owing to the size and complexity of most system designs.

The wide variety of data base constituents, attributes, and relationships offers a great deal of flexibility and power in CADSAT. Among the kinds of statements used to describe properties of data base items are:

- o Identification and description statements that are common to all names; these specify SYNONYMS, ATTRIBUTES, DESCRIPTIONS, MEMOS associated with the name, keywords, etc.

- o Flow statements that relate data and processes such as RECEIVES, USES, DERIVES, etc.

- o Size and volume statements that indicate CARDINALITY and VALUES

CADSAT has many report options that provide output pertaining to the data objects or relationships. These reports present information at several levels of detail and completeness. The forms the reports take include matrices, tree structures, graphs, and text structured by appropriate indentation and spacing.

Reference:

Moore, C. R., User Requirements Analyzer Version 2.0 User's Manual for IBM 370/158/OS/TSO Installation, Released by ISTAO, Electronic Systems Division, Air Force Systems Command.

CLARKE

Clarke's Program Testing System symbolically executes ANSI FORTRAN program paths, creates symbolic representation of outputs, and provides a system of inequalities representing the conditions under which the path is executed. Six types of possible data-dependent errors are simulated in an attempt to reveal sets of data which would cause execution errors.

The system is written in ANSI FORTRAN and uses the portable ALTRAN system for symbolic algebraic manipulations. Selected program paths are analyzed in three phases: 1) symbolic execution, 2) inequality simplification, and 3) inequality solution.

When an input statement is encountered during symbolic execution, two arrays, I and X, are used to represent input values (e.g., I(4) represents the fourth input value, which is integer). When an output statement is encountered, the symbolic expression corresponding to the value to be output is provided. At each conditional transfer, an inequality is generated to correspond to control flow for the specified path. An attempt is made to evaluate this inequality to true or false. If this is not possible, it is added to the previously generated inequalities. In Phase 2, ALTRAN is used to simplify these inequalities. Phase 3 is used to solve the inequalities if they are linear. The inequalities are supplied one at a time to the solver, which checks to see if the previous solution satisfies the new inequality. If not, an attempt is made to find a new solution and proceed with the next inequality.

Language-dependent properties, such as integer truncation, are represented by additional inequalities necessary for the validity of generated test data.

The system attempts to generate artificial constraints that simulate data-dependent errors. These inequalities are temporarily added to those to be considered by the inequality solver. If this modified set of inequalities has a solution, a potential run-time error exists. In any event, the temporary addition is removed and processing continues. This process is used for six potential errors:

- o Division by 0
- o Subscripts out of bounds
- o Illegal variable dimensions
- o Illegal DO parameters
- o Illegal mixed-mode expressions
- o Reference to undefined variables

<u>References</u>:

Brown, W. S., <u>Altran User's Manual</u>, Vol. 1, Bell Telephone Lab, 1973.

Clarke, L. A., <u>A System To Generate Test Data and Symbolically Execute Programs</u>, Univ. of Colorado, Rept. CU-CS-060-75, Feb 1975.

Clarke, L. A., "Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation," Ph.D. Thesis, Univ. of Colo. 1976.

Clarke, L. A., "A Program Testing System," <u>Proc. of the Annual Conf. of the ACM</u>, Houston, TX, 20-22 Oct 1976.

CWRU

The Case Western Reserve University (CWRU) Program Verification System uses the inductive assertion method of Floyd to verify programs. It processes PASCAL programs to which intermediate assertions have been added.

The system consists of three main components: 1) verification condition generator (VCG), 2) subgoal generator (SG), and 3) subgoal solver (RES).

The VCG processes a PASCAL program augmented with entry/exit conditions and intermediate assertions and uses a dictionary to provide a macro scheme for these assertions. Predicates defined in dictionary entries are also later used by the SG. VCG handles array references by using the 'acc' and 'chg' functions of McCarthy and Painter. The standard PASCAL notation in the program is translated by VCG to expressions using these functions. The output of VCG is a set of verification conditions.

SG first splits the verification conditions into subgoals, each of which is a terminal node of an AND/OR tree. This involves conversion of arithmetic expressions to normal form, substitution of dictionary-defined predicates, elimination of all logical connectives other than AND, OR, and NOT, and substitutions corresponding to predicates of the form X=Y. Next, SG attempts to solve each subgoal using further simplification and restricted resolution. If this fails, SG attempts to use high-level properties of the dictionary entries to solve the subgoal. SG is useful in solving many subgoals which are tedious and straightforward. It is capable of solving most subgoals, but if it fails RES is used to prove the subgoal. RES is a resolution theorem prover which makes efficient deductions in a theory of total ordering.

References:

Hookway, R. J., A Program Verification System, Report No. 1171, Jennings Computing Center, Case Western Reserve University, Jan 1976.

Hookway, R. J., and G. W. Ernst, "A Program Verification System," Proc. of the Annual Conf. of the ACM, Houston, TX, 20-22 Oct 1976.

McCarthy, P., and J. A. Painter, "Correctness of a Compiler for Arithmetic Expressions," Proc. of the Symp. on Appl. Math., American Math Soc., Vol. 19, 1967.

DAVE

DAVE is used to analyze FORTRAN programs for data flow anomalies. It creates a flowgraph for each input routine and a callgraph for the entire input program, indicating which routines call what other routines. Associated with each program statement is a list of variables appearing in that statement, and whether each is referenced, defined, or undefined (a variable is undefined upon exit from a block in which it was declared; DO-loop control variables are undefined upon exit from the loop). With this information, together with the model of the program's control flow, DAVE detects the following anomalies:

- o All situations in which an undefined variable is referenced
- o Most situation in which a defined variable is redefined
- o Most situations in which a defined variable is undefined

Analysis is performed through routine boundaries by processing routines in a leaf-upward order in the callgraph, using a variable-by-variable depth-first search.

The DAVE system has the following limitations:

- o Since DAVE performs static flow analysis, it cannot determine which element of an array is being referenced when the subscripts are not constants. As a result, DAVE treats all arrays as simple variables, which diminishes its anomaly-detection power.

- o It is possible to write a program in which each of two subroutines contains a reference to the other, but no execution would cause recursion. This obscures the determination of leaf nodes in the callgraph, and DAVE does not adequately handle this situation.

- o DAVE assumes that all parameters of a routine represent different variables. If this is not the case, incorrect analysis may be performed.

The prototype DAVE system processes routines at the rate of about two to three statements per second on the CDC 6400. It has been used on several production programs; most of the anomalies discovered were of the first two kinds described above.

References:

Fosdick, L. D., and L. J. Osterweil, "DAVE - A FORTRAN Program Analysis System," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

Fosdick, L. D., and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys, Vol. 8, No. 3, Sep 1976.

DISSECT

The DISSECT system is used to analyze FORTRAN programs. It performs symbolic execution and provides symbolic path constraints, allowing the user considerable flexibility in determining what analysis is to take place.

The user supplies a number of CASEs indicating the types of analysis to be performed. Each CASE consists of a text description, a list of output variables to be printed, and a path description. The path description may represent a single path, no paths, or a set of paths. The path description may consist of general commands (e.g., select all consistent branches, execute all loops at most once) and specific requests about branch points and loops (e.g., take the greater-than branch at statement 10, execute the loop at statement 15 four times). These control commands may be nested within IF-THEN-ELSE-type constructs for increased flexibility. For each path represented in the path description, DISSECT can print the statement numbers occurring in the path, the symbolic predicate corresponding to the path, and the symbolic values for variables requested by the user.

The user may initialize input variables with numeric or symbolic values. Intermediate symbolic values of variables and the path constraints may be output during execution of the program.

The output provided by DISSECT can be used to generate test data, prove correctness, or compare a program to its specifications.

References:

Howden, W. E., "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, Vol. C-24, May 1975.

Howden, W. E., "Automated Program Validation Analysis," Proc. of the Fourth Texas Conf. on Computing Systems, 17-18 Nov 1975.

Howden, W. E., "The DISSECT Symbolic Evaluation System," Proc. of the 1976 National Comp. Conf., New York, NY, 6-10 Jun 1976.

Howden, W. E., Symbolic Testing - Design Techniques, Costs and Effectiveness, NTIS PB268-517.

Howden, W. E. and J. Laub, "Automatic Case Analysis of Programs," Proc. of the Comp. Sci. and Statistics: Eighth Annual Symp. on the Interface, Los Angeles, CA, 13 Feb 1975.

EFFIGY

EFFIGY, whose development has been under way since 1973, is used for debugging and testing using symbolic execution. The input language is a PL/I-type language including:

- o External procedures and PL/I parameter passing
- o Integers and arrays of integers as the only data types
- o A typical set of arithmetic, relational, and logical operators
- o Assignment statements, IF, DO-WHILE, GOTO, iterative DO, simple READ and WRITE, and component statements

Among the facilities offered by EFFIGY are:

- o Debugging and testing with symbolic or normal program execution
- o Test management providing for exploration of the symbolic execution tree
- o Automatic checking of test case results against user-supplied assertions
- o Program verification for generating verification conditions from assertions and symbolic execution

Interactive debugging with EFFIGY allows the user to follow an execution trace consisting of statement number, source statement, computational results, or any combination of these. Breakpoints may be inserted to interrupt execution and allow the user to examine and set variables before resuming execution.

Execution of a program is begun by invoking the routine with numeric and/or symbolic parameters. Symbolic execution proceeds, building the path condition and keeping track of the symbolic values of variables, until a branch is reached which EFFIGY cannot resolve. The user is then allowed to add a predicate to the path condition and have EFFIGY attempt to reevaluate the branch condition, or force the system to take a user-specified branch. The user may also save the state of execution in order to return later and select a different branch. The TEST facility instructs EFFIGY to save and restore execution states automatically so that all paths of the execution tree are considered. An upper bound to the number of statements to execute must be supplied to prevent an attempt to exhaustively search an infinite tree.

Correctness proofs of individual paths are accomplished by providing an initial ASSERT statement describing the input variables, supplying a PROVE statement describing the end state of execution, and instructing EFFIGY to use its own controller to enumerate paths and force choices at uresolvable branch points.

<u>References</u>:

King, J. C., "A Program Verifier," <u>Proc. IFIP Cong. 71</u>, North-Holland, Amsterdam, 1971.

King, J. C., "Proving Programs To Be Correct," <u>IEEE Transactions on Computers</u>, C-20, Nov 1971.

King, J. C., "A New Approach to Program Testing," <u>Proc. of the Int. Conf. on Reliable Software</u>, Los Angeles, CA, 21-23 Apr 1975.

King, J. C., "Symbolic Execution and Program Testing," <u>Communications of the ACM</u>, Vol. 19, No. 7, Jul 1976.

King, J. C. and R. W. Floyd, "An Interpretation Oriented Theorem Prover Over Integers," <u>J. Computer Syst. Sci.</u> 6 Aug 1972.

<u>FACES</u>

The FORTRAN Automatic Code Evaluation System (FACES) is designed to assist in the development, testing, and maintenance of ANSI FORTRAN programs. It consists of two parts, one which generates a data base model of the FORTRAN program, and one which processes user commands to analyze the data base and report requested static characteristics of the program.

The data base consists of three tables:

- o Symbol table, containing all symbolic elements of the input program
- o Use table, indicating how and where each symbolic element is referenced
- o Node table, recording the program flow structure

The second part of FACES provides for the handling of three types of requests: software quality checks, specific user requests, and documentation requests. Software quality checks include the identification of error-prone constructs (e.g., using a constant as an actual parameter to a subroutine), interface checks (consistent COMMON definitions and subroutine parameters), program structure tests, coding standards, and uninitialized variable checks. Specific user requests may be issued to retrieve information about specific variables, statements, or routines, and their influence on other entities of the program. Documentation requests may request production of various cross-reference tables and program graphs.

FACES is written in ANSI FORTRAN and is now operational on the Univac 1108 and CDC 6400. Current table sizes limit the input program to a maximum of 200 modules, each with less than 700 statements.

Reference:

Ramamoorthy, C. V., and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, Mar 1975.

FADEBUG-1

The FADEBUG-1 system is designed to be used in the early debugging stages of program development. It processes programs written in assembly language and consists of two basic functions:

- o Automatic listing of all physical program paths of a module from entry to exit. This is to aid the programmer in checking out all logical program paths.

- o Comparison of user-supplied data with memory contents after execution. FADEBUG initializes registers and memory on the basis of SET statements provided by the user, then executes the program and checks the contents of memory for correct values on the basis of CHECK statements. Inconsistencies are printed.

Path listing is restricted in the following ways:

- o Paths which pass the same node more than three times are not listed.

- o No more than 200 paths are listed.

- o A path reaching a branch-by-register or indirect branch is considered to terminate at that point.

Experimentation has shown that by using the system, manpower can be decreased by 11% and debugging speed improved by 17% at the module test stage of program development.

Reference:

Itoh, D., and T. Izutani, "FADEBUG-1, A New Tool for Program Debugging," IEEE Symp. on Computer Software Reliability, 1973.

PET

The Program Evaluator and Tester (PET) is a tool that automatically generates self-metric software from existing software and provides static and execution statistics for program runs. The tool analyzes the source code and outputs the source modified with code to collect statistics on: execution counts; minimum, maximum, first, and last values assigned to variables in nontrivial assignment statements; and minimum, maximum, first, and last values assigned to DO-loop control variables. The user may specify any combination of these three modifications.

After the instrumented source is executed, the following outputs are available:

- o Number and percentage of all executable source statements in the part executed
- o Number and percentage of branches taken
- o Number and percentage of subroutine calls executed
- o Number of times each subroutine was called
- o Subroutine timing
- o Specific data associated with each executable statement:
  - *Execution counts*
  - Branch counts
  - Data range values on assignments
  - Ranges of DO-loop control variables

Versions of PET exist for FORTRAN, Metatranslator of McDonnell Douglas, and Metaassembler for NASA's SUMC computer line.

The PET preprocessor, which instruments the source code, is written in Metatranslator. The postprocessor, which generates the reports, is to be written in the same language as the source. A FORTRAN version has been built and a COBOL version has been specified.

Reference:

Stucki, L. G., "Automatic Generation of Self-Metric Software," IEEE Symp. on Computer Software Reliability, 1973.

SDVS

The Software Design and Verification System (SDVS) is an integrated collection of software tools used to support the Digital Avionics Information System (DAIS). SDVS is used to aid the development, coding, and testing of DAIS software. It offers the following services:

- File Configuration: All files are kept in a version/revision fashion. All revisions are cataloged as unique records in a "difference file." As a result, the system may obtain any version by combining the appropriate differences. Commands are available for creating, editing, accessing, outputting, and compiling files. Source and test case files may be processed.

- Simulation: SDVS will automatically set up and submit a simulation run based on a user-supplied test case file. Operator interactions, file retrieval, and execution are handled by SDVS. The post-run processing may optionally be performed by SDVS.

- Post-Run Edit: A user-supplied directives file is used to determine what routines and parameters are to be used to analyze the output tape.

- Rollback: The user is allowed to rerun a previous simulation from a point as stored on a snapshot tape. Changes may be made to the test file to provide further analysis.

- Supervisor Services: The capability to delete files and set programmer-file access lists is available only to the SDVS manager.

SDVS has been used by both Air Force and contractor personnel to assist in the design, implementation, and testing of mission software for DAIS.

Reference:

Hollowich, M. E., and M. G. McClimens, Software Design and Verification System, Air Force Avionics Laboratory Tech. Rept. AFAL-TR-76-200, AD A042126.

STRUCT

The Structured Programming Verifier (STRUCT) was developed at TRW to verify automatically that programs follow the restrictions of structured programming. The basic system audits code for proper use of the SEQUENCE, IF-THEN-ELSE, DO-WHILE, DO-UNTIL, and CASE program constructs.

The system first translates an input FORTRAN program into a directed graph, where the nodes represent straight-line segments of code and the arcs represent transfers of control. Using a set of graph-reduction rules, this model is reduced to either: 1) a single node, indicating that the input program adhered to the structured programming constructs, or 2) a graph of more than one node, indicating that the input program was not well structured. This final state is the output of STRUCT.

The original five constructs were found to be too restrictive in that the use of only these constructs results in overly large and slow-executing programs. As a result, STRUCT was modified to allow for three other structures: 1) the escape mechanism, for early termination of a loop, 2) error processing termination, and 3) multiple entry/exit. The algorithm attempts to obey the original reduction rules as long as possible. When no other reduction is available, one application of a reduction rule corresponding to the above three modifications is made; then processing continues with the original rules again.

The STRUCT system has been operational since May 1974 and is used about 200 times per month. It is capable of processing 7000 source statements of 150 routines in 8 seconds of CPU time on the CDC 7600.

Reference:

Brown, J. R., and K. F. Fischer, "A Graph Theoretic Approach to the Verification of Program Structures," Third Int. Conf. on Software Engineering, 10-12 May 1978.

TPL/F AND TPL/2.0

The FORTRAN Test Procedure Language (TPL/F), developed at General Electric, aids in testing FORTRAN programs. The language is used to define input arguments to routines, output values expected from the routines, and directions on how the system is to simulate called subroutines and I/O devices. The test cases so defined are then executed, and the system reports on inconsistencies between the program's behavior and the specified values.

Input values are specified by supplying a subroutine name, formal parameter name, and the value the parameter is to assume upon entry. Output values are specified by supplying a Boolean expression over the output variables, indicating the value each variable should have upon exit from the routine.

Called subroutines are simulated by supplying a FORTRAN-like subroutine in which the user can access the system variable #TEST, indicating the test case being executed. The user can also employ the ABORT statement, which if executed instructs the system to print an appropriate message, terminate that test case, and continue with the next test case defined in the test procedure.

The I/O simulators are defined by supplying a unit number and a list of records to be used as input or output. Input records are read in during execution and output records are used to verify the program's output.

To prevent lengthy test procedures, TPL/F provides a macroprocessor enabling the user to compress the description of many test cases of similar structure. TPL/F has the advantage that the internal structure can be used to design test cases. Local variables as well as parameters may be used to verify intermediate results, and the user may specify that only a section of the code is to be executed.

The TPL/2.0 test language, a direct outgrowth of TPL/F, offers two main capabilities over its predecessor: 1) the language to describe test cases allows very compact procedures to be specified, and 2) the REVISE command can be used to simplify the initial test procedure coding and to automate later test revisions. The REVISE command executes a test procedure as in TPL/F but with the following difference. The output values are not verified against the test procedure outputs, but are instead used to generate a new test procedure in which the actual outputs are used as the test procedure outputs. This allows the user to generate initial test procedures easily by first inspecting the outputs and then, if desired, using REVISE to create a test procedure which specifies those outputs as values to be verified. This procedure may also be used to update a test procedure when a source module is updated.

References:

Panzl, D. J., "Test Procedures: A New Approach to Software Verification," Second Int. Conf. on Software Engineering, 13-15 Oct 1976, pp. 477-485.

Panzl, D. J., "Automatic Software Test Drivers," Computer, Apr 1978.

Panzl, D. J., "Automatic Revision of Formal Test Procedures," Third Int. Conf. on Software Engineering, 10-12 May 1978.

APPENDIX B
ABSTRACTS OF CURRENT MICROPROGRAMMING TOOLS

A good deal of recent microprogramming literature was reviewed to identify tools that were specifically oriented toward microprogram verification and validation. The relatively small number of abstracts in this appendix reflects the general scarcity of information in this area. The following sources provided valuable background information in the related area of microcode development tools.

Dasgupta, S., and J. Tartar, "The Identification of Maximal Parallelism in Straight-Line Microprograms," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 986-992.

De Witt, D. J., M. S. Schlansker, and D. E. Atkins, "A Microprogramming Language for the B-1726," Conf. Record of the Sixth Annual Workshop on Microprogramming, College Park, MD, 24-25 Sep 1973, pp. 21-39.

Evans, Moffett, and Merwin, "Design of Assembly Level Lanuguage for Horizontal Encoded Microprogrammed Control Unit," Conf. Record of the Seventh Annual Workshop on Microprogramming, Palo Alto, CA, 30 Sep - 2 Oct 1974, pp. 217-224.

Ramamoorthy, C. V., et al., "A Higher Level Language for Microprogramming," Conf. Record of the Sixth Annual Workshop on Microprogramming,College Park, MD, 24-25 Sep 1973, pp. 139-144.

SMITE Installation and Analysis: SMITE Training Manual, TRW Defense and Space Systems Group 30417-6002-RU-00, RADC-TR-77-364, 12 Aug 1977.

Tsuchiya, M., and M. Gonzales, "Toward Optimization of Horizontal Microprograms," IEEE Transactions on Computers, Vol C-25, No. 10, Oct 1976, pp. 992-999.

The tools abstracted in this appendix address a subset of the tool classifications from Appendix A as follows:

o Abort Diagnosis Tools	BFTSC ICS, Interactive Debugger, PRIM Debugger
o Breakpoint Controllers	BFTSC ICS, Interactive Debugger, PRIM Debugger
o Simulators/Symbolic Executors	MCS
o Formal Verifiers	MCS, STRUM

The abstracts included here were written under the same assumption identified in Appendix A, namely, that the reader is familiar with the terminology common to verification and validation tools.

BFTSC ICS

The Brassboard Fault-Tolerant Spaceborne Computer (BFTSC) Microcode Interpretive Computer Simulator (ICS) is a tool used to validate the microprogram of the Brassboard Fault-Tolerant Spaceborne Computer built for the Air Force Space and Missile Systems Organization. It is a FORTRAN program, developed by Logicon on a CDC 6700 computer. It simulates the hardware of the BFTSC at the microinstruction execution level.

The BFTSC ICS provides diagnostic capabilities normally unavailable on hardware. Diagnostic information includes:

- A microinstruction trace which optionally occurs after the execution of each microinstruction; the trace includes the values of all flip-flops and registers, the microinstruction address, the end condition, and the symbolic ALU function being performed
- A macroinstruction trace
- A microinstruction address trace summary for each macroinstruction executed

Inputs include the BFTSC microprogram, an assembled macroinstruction test program, and time- and location-keyed trace commands.

The program is organized as a set of program segments which simulate either data flow or operations within the CPU. Each program segment represents a particular time slot (e.g., microinstruction fetch, microinstruction decoding, operand fetch) occurring during the execution of a microinstruction. Every program segment is executed once for each microinstruction executed.

The BFTSC ICS was designed to execute a small number of test cases. Development time and maintenance costs were of greater concern than execution time efficiency. It runs on the order of 2500 times slower than real time.

The BFTSC ICS is a tool used in conjunction with, rather than as a replacement for, manual analysis. Regardless of execution speed, the large number of instruction and operand combinations makes exhaustive automated testing impractical. Suitable macroinstruction programs must be constructed to ensure that all microcode branches are exercised. In testing the BFTSC microcode, manual code analysis accounted for 24% of the errors found, while 67% were detected through code execution on the BFTSC ICS and 9% were found while executing on the hardware.

Reference:

Wheatley, R., "A Simulation Technique in Microprogram Validation," Proc. of AIAA Computers in Aerospace Conference, Los Angeles, CA, 31 Oct - 2 Nov 1977.

INTERACTIVE DEBUGGER

The Interactive Debugger for software and firmware on an Intercomputer i-50 minicomputer was developed by the MITRE Corporation. Its hardware is organized in a dual-processor configuration, using two identical Intercomputer i-50 minicomputers sharing a common memory. Its software runs on one processor, using standard i-50 firmware, and the firmware and software to be debugged run on the other processor.

The dual-processor configuration has several advantages:

- o The program being debugged can run in real time.
- o The debugger can run on debugged firmware.
- o The debugger can halt the second processor at any time, even if no breakpoint is reached, saving the full state of the second processor.

The following firmware debugging requests are provided:

- o Examine and Modify: Examine a single location (or a set of contiguous locations) in control store, or examine a register. Print the contents of the location as a hexadecimal number or as an instruction. Insert a new value at the specified location if desired.
- o Set Breakpoint: Set a breakpoint at the specified control store address. This command must be used in conjunction with the Next Instruction command; otherwise the processor will continue on to the next instruction. Only one breakpoint may ever be active.
- o Clear Breakpoint: Clear the breakpoint in control store.
- o Next Instruction: Execute the macroinstruction at the current or specified location and halt.
- o Go: Start the processor at the current or specified location in macroprogram store.

There are several problems and limitations with this approach to software and firmware debugging:

- o Setting a breakpoint in firmware requires the replacement of a microinstruction.
- o Since the processors share a common memory and memory protection is under firmware control, it is not possible to protect the debugger's software from the second processor.
- o A small amount of microcode must be added to the firmware being debugged.
- o The cost of a dual-CPU system to debug software for a single processor may be prohibitive. The approach was used in this case partly because the debugged dual Intercomputer i-50 system would eventually be used as a dual-processor system.

The debugger's performance is described as "very satisfying." Most of the on-line debugging time was spent using the debugger. During this time, access to the operator's console was minimal.

Reference:

Gasser, M., "An Interactive Debugger for Software and Firmware," Conf. Record of the Sixth Annual Workshop on Microprogramming, College Park, MD, 24-25 Sep 1973, pp. 113-119.

MCS

The Microprogram Certification System (MCS) is an interactive system used in the proof of microprogram correctness. The system was developed by Leeman, Joyner, and Carter at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York. It is written in LISP.

MCS provides a mathematical proof that machine instructions are properly implemented by microcode. The computer whose microcode is being verified is formally defined at both the machine instruction and microinstruction levels, using an extended version of the Vienna Definition Language (VDL). MCS then attempts a formal, interactive proof of microprogram correctness by symbolically executing machine instructions and corresponding microinstructions, generating theorems that must be true for the microcode to properly implement the machine-level instructions, and proving the theorems.

MCS is organized in five segments:

- o Supervisor: Enables the user to observe and control the proof. Other segments are invoked by the supervisor using a set of standard commands.

- Path Tracer: Symbolically executes a machine instruction and corresponding microinstructions, using the input computer descriptions. Determines which paths will be executed under all possible conditions.

- Simplifier: Simplifies symbolic expressions involving APL and logical operators (e.g., (-exp) + exp simplifies to zero).

- Theorem Generator: Generates a set of theorems whose proof is necessary to establish that the machine instructions and the corresponding microinstructions compute the same result.

- Theorem Prover: Reduces theorems through case or conjunction splitting under user control. Attempts to simplify theorems to true.

An attempt at verifying an emulation of the IBM 360 on the IBM Hybrid Technology Computer revealed problems related to size and complexity. The simulation relation for the emulation was only partially formulated, and the emulation was only partially verified. The slow symbolic execution necessitated optimization, which required a good knowledge of both the microcode and data flow.

The MCS found four errors in the microcode for the S-machine, a simple hypothetical computer used by IBM for teaching microprogramming. The first involved an incorrect number of shifts in the left-shift and right-shift instructions. The second involved the execution of the stack-to-index instruction where add-stack-to-index or subtract-stack-from-index were specified. A third error was found in the instruction fetch microcode. The fourth error was a failure to mask certain bits to zero during address generation.

It must be remembered that the MCS verifies microprograms with respect to the machine specifications written in VDL and with respect to a stated simulation relation. Errors may exist in microprograms "proved" correct by the MCS.

References:

Birman, A., "On Proving Correctness of Microprograms," IBM Journal of Research and Development, May 1974, pp. 250-266.

Joyner, W. H., et al., Automated Verification of Microprograms, IBM Research Report FC 5941 (#25780), 12 Apr 1976.

Leeman, G. B., et al., An Automated Proof of Microprogram Correctness, IBM Research Report RC 6587 (#28423), 20 Jun 1977.

PRIM DEBUGGER

The Programming Research Instrument (PRIM) is an interactive microprogrammable environment developed by USC Information Sciences Institute. It is available to remote users through the ARPANET, providing an integrated set of programming aids to create, debug, and execute programs for experimental computer environments. The TENEX time-sharing system, under which PRIM runs, provides access to editors and compilers for creating emulators. PRIM also provides an environment for configuring and debugging target systems that can be operated by the user in the language of the original target system. PRIM has already been used to emulate the following target computer systems: U1050, UYK-20, and the Intel 8080 (chip).

The PRIM system consists of an MLP-900 microprogrammable processor and appropriate software: PRIM Exec, PRIM Debugger, GPM (general-purpose microprogramming language) compiler, and TENEX.

The PRIM Debugger is a table-driven, target-machine-independent interactive program for debugging a PRIM emulator or a target program running on such an emulator. It is tailored to a specific target machine by tables prepared as part of an emulation tool. It permits the user to set and clear breakpoints and to examine, modify, and monitor target system locations. Target system assembly language and symbolic names are recognized, and arithmetic is performed according to the conventions of the target machine.

Breakpoints can be set on events (predefined conditions) or location references. The debugger may contain a break-time "program" to be executed when the break condition is met. If the break-time program ends without a GO command or if no break-time program exists at the break, a message describing the break is given and the command level is entered. Otherwise execution is automatically resumed; the user receives no indication that a breakpoint occurred unless the break itself produced output.

The advantages of the PRIM debugger are as follows:

- o Vocabulary and commands can be easily tailored to a specific machine or research application.
- o The debugger provides better user debugging facilities than the actual machine, while still producing bit-compatible results.
- o The debugger command language includes feedback and help if needed.
- o The debugger allows a complete checkpoint and subsequent restoration of the state of emulation.

A limitation is that the MLP-900 is no longer manufactured, so a potential user is constrained to the ISI system and cannot acquire one of his own.

References:

Britt, B., et al., PRIM System: Overview, ARPA Order No. 2223, ISI/RR-77-58, Information Sciences Institute, Mar 1977.

Gallenson, L., et al., PRIM System: AN/UYK-20 User Guide, User Reference Manual, ARPA Order No. 2223, ISI/TM-77-5, Information Sciences Institute, Oct 1977.

Gallenson, L., et al., PRIM System: Tool Builder's Manual, User Reference Manual, ARPA Order No. 2223, ISI/TM-78-7, Information Sciences Institute, Jan 1978.

STRUM

The Structured Microprogramming (STRUM) language is a vertical, high-level microprogramming language. STRUM was implemented at UCLA by David Patterson on an IBM 360/91. It produces object code for the Burroughs D-machine. STRUM was written in XPL, a PL/I dialect, and REDUCE, an algebraic manipulation language based on LISP. STRUM has been successfully used in the coding and formal verification of an emulator for the HP-2115.

The language has several convenient programming features. These features include macros, procedures, and structured programming control structures. The PASCAL-like syntax includes the reserved words WHILE, REPEAT, FOR, LOOP, EXIT, CASE, BEGIN, END, IF, THEN, and ELSE.

STRUM also has facilities for formal program verification. Assertions describing the state of program variables are inserted by the programmer at critical points in the source code. The STRUM compiler produces theorems from the source code and assertions which, if true, show that the STRUM program is consistent with its assertions. The theorems are processed by a simplifier and an interactive theorem prover.

Although the program assertion method aids in the verification of programs, it cannot guarantee correctness. STRUM cannot prove programs correct: it determines whether assertions about programs are consistent with the programs. The method is useless if output assertions do not describe the intent of the program. Development of the assertions may be difficult.

The STRUM compiler employs a simple three-part optimization strategy. First, IF-THEN-FI statements are compiled into a single microinstruction. Next, pairs of microinstructions are examined and combined into a single microinstruction if possible. Third, chained branches are removed from object microcode. These optimizations reduced the size of the HP-2115 emulator by 20%. In addition, the language includes a parallel statement which allows the user to explicitly state operations that can be performed in parallel.

A STRUM emulation of the HP-2115 minicomputer uses fewer microinstructions than an emulator written in TRANSLANG, the assembler for the Burroughs D-machine. The STRUM emulator is also one-third faster than the assembly language version.

Verifying the code for the HP-2115 emulator produced 1700 pages of theorems and was completed in 3 days. Ten errors were found in the source code, 10 were found in the assertions, and 11 were found in the specifications for the emulator. The amount of output and the verification time compared favorably with the traditional snapshot-dump and front-panel-toggling techniques used to debug the assembly language emulation.

STRUM was expensive to develop. Patterson estimates that production versions of an efficient optimizing compiler and verification system would require 5 to 10 man-years to develop.

References:

Patterson, D. A., "Verification of Microprograms," Ph.D. Thesis, Department of Computer Science, UCLA, 1976.

Patterson, D. A., "STRUM: Structured Microprogram Development System for Correct Firmware," IEEE Transactions on Computers, Vol. C-25, No. 10, Oct 1976, pp. 974-985.

APPENDIX C
ANALYSIS OF CURRENT SOFTWARE TOOLS

This appendix identifies features of current software tools that are not available in current microprogramming tools, and identifies the microprogramming problem areas addressed by each tool.

ACES

The significant feature of ACES not available from current microprogramming tools is the ability to monitor assignments to specified variables and to detect violations of specified upper and lower bounds. This feature applies directly to incorrect accessing or storing of data and to errors in equation computation and arithmetic.

Other ACES features, while applicable to microprogram development, are standard features of high-level language compilers. A cross-reference feature provides or verifies documentation. The identification of undefined labels, unreferenced labels, unreachable statements, terminating statements, and program loops applies to branching errors, logic and sequence errors, and disagreement between code and detailed design specification.

While variable monitoring is applicable to microprograms, code insertion to accomplish it is impractical owing to timing and control-store size constraints. Variable monitoring is more appropriately performed by microcode simulators.

AIDS

The capabilities of the AIDS system closely resemble those of the BFTSC ICS, a current microprogramming tool. Both provide the ability to print registers or variables based on location references. This ability, which can be used to provide a complete execution trace, is applicable to a large number of problem areas, including incorrect accessing or storing of data, branching errors, logic and sequence errors, and errors in equation computation or arithmetic.

One significant feature of AIDS, compared to the BFTSC ICS, is that AIDS is interactive, where the BFTSC ICS is batch oriented. The ability of the user to modify the system's execution according to intermediate results is particularly useful since simulation is very slow.

Another feature, the ability to change from simulation to direct execution at specified locations, is useful but less applicable to microprogramming. Direct execution may be as much as 2000 times faster than simulation, but it can only be used if the host machine for the simulator is also the target machine for the microcode.

AMPIC

AMPIC provides several capabilities not available from current microprogramming tools. These capabilities--flowcharting, flowchart restructuring, timing analysis, symbolic execution, and decompilation--are applicable to a number of microprogramming problem areas.

Flowcharting and flowchart restructuring are adaptable to microprogramming with the establishment of conventions for representing parallel operations. Flowcharts produced from source code can identify disagreement between code and detailed design specification, branching errors, logic and sequence errors, and incomplete or erroneous documentation. Restructuring can reveal branching errors.

Timing analysis is another useful capability. The BFTSC ICS provides execution times only for specified paths. AMPIC determines minimum and maximum execution times for entire routines or groups of routines. This timing facility applies to errors in timing and process allocation and to errors in interruptibility and data coherency.

Symbolic execution and decompilation are also applicable to microprogramming. They identify incorrect accessing or storing of data, errors in equation computation or arithmetic, and disagreement between code and detailed design specification. Although formal verifiers such as STRUM provide more powerful verification capabilities, symbolic executors and decompilers may have superior cost/performance characteristics.

ATDG

ATDG offers a significant improvement over current methods of generating test data for microprograms. The BFTSC ICS, which uses a test program to exercise all microcode branches, relies on manual methods to produce the test program. The slow execution speed of computer simulators makes particularly attractive ATDG's objective of generating the fewest number of paths required to collectively exercise all predicate outcomes and verify the executability of these paths.

Static error analysis capabilities resembling DAVE's are discussed in the evaluation for DAVE.

AUT, FADEBUG, AND TPL

The AUT, FADEBUG, and TPL tools have little to offer over the capabilities of an interpretive simulator. They are primarily automated aids used to compare the end results of a user-supplied test run with expected end results also supplied by the user. Data flow across routine boundaries during a test run may also be compared to expected values. No analysis other than this "end-to-end" comparison for routines is provided. The likelihood of uncovering errors in this manner is therefore completely dependent upon the user's insight into selection of test case data sets.

It is conceivable that this type of analysis may aid the verification of code vs. specification on the routine level. However, since numeric results are used in the comparison, such a verification would require exhaustive testing, which is usually unrealistic if not impossible.

It should be noted that these tools indirectly provide for the documentation and reusability of test cases, by virtue of the fact that the test cases supplied to the tools must be well defined and formatted.

BAP AND PET

The features found in BAP and PET but not available in current microprogramming tools include various embellishments of execution monitoring. In particular, identifying the minimum and maximum values assigned to selected variables, revealing execution counts for statements and branches, and the ability to collect these statistics over several test runs aid in discovering errors in the problem areas of incorrect accessing and storing of data, branching errors, errors in equation computation or arithmetic, and logic and sequence errors.

These embellishments may be considered as doing nothing more than automatically providing a compact representation of the analysis of execution trace data. Although such traces are available in current microprogramming tools, the analysis equivalent to the reports provided by BAP and PET is not, and would be a significant additon.

CADSAT

The capability of CADSAT to aid in creating, analyzing, and maintaining a data base of user requirements is not available in any of the current microprogramming tools. The problem area that it addresses is incomplete or erroneous specifications. Although MCS and STRUM are able to detect inconsistencies in specifications through the use of their formal verification capabilities, CADSAT is unique in providing semantic and syntactic checks of each requirement, data management capabilities, contextual checks to guarantee consistency, and extensive analysis of the relationships among the requirements in the data base. These capabilities are applicable to microprogram as well as software development efforts.

CLARKE, DISSECT, AND EFFIGY

The symbolic execution facilities provided by CLARKE, DISSECT, and EFFIGY assist in detecting errors in several categories. Symbolic execution may be used to reconstruct arithmetic formulas from sequences of instructions and provide precise descriptions of the cases handled by the source code. These provisions may be used to discover disagreement between code and detailed design specification and logic and sequence errors. These tools also offer varying degrees of automated path selection. During automated path selection the case-analysis and case-description capabilities are useful for finding branching errors.

A symbolic execution system has the ability to search explicitly for possible occurrences of specific errors such as division by zero or subscripts not within the range of the array bounds. Possible truncation effects may also be modeled within the predicates describing paths and/or output values of variables. These capabilities may help to display errors in equation computation or arithmetic and incorrect accessing or storing of data.

Although symbolic execution applies to several problem areas, the use of CLARKE, DISSECT, and EFFIGY offers no significant improvement over current microprogramming tools, since microprogram verifiers do exist. A verifier essentially "guarantees" that the code is correct if the user-supplied assertions are correct, but symbolic execution systems need no such critical information from the user. This is one important difference which suggests that symbolic evaluators should not be completely overshadowed by program verification systems.

CRWU

The CWRU Program Verification System uses the same inductive assertion method used by STRUM, a current microprogramming tool. For this reason the CWRU system does not offer an improvement over current methods.

DAVE

The DAVE system offers error-detection capabilities primarily in the area of disagreement between code and detailed design specification. The data flow analysis performed at the routine level assists in verifying proper sequencing of variable assignments. On the global level, data flow analysis can assist in verifying routine interfaces by indicating the routines that reference or define each variables. This information could be compared with a program specification to assure proper routine communication.

The DAVE system also contributes to the area of incomplete or erroneous documentation by providing a global callgraph that indicates the dynamic routine call structure for the set of routines analyzed.

Adapting the concepts involved in the DAVE system to the processing of microcode would not present any major problems. Data flow takes place primarily between registers and memory instead of between variables.

FACES

Software quality checking is a FACES capability that is applicable to microprogramming. The functions identified with software quality checking in the FACES abstract address the problem areas of violation of programming practices and incorrect access and storing of data.

Automated software quality checking is not a capability of current microprogramming tools. Programming practice violations are currently addressed only by the syntactic checking and improved readability offered by high-level microprogramming language compilers.

FACES cross-referencing is not new to microprogramming. Cross-references are expected features of microprogramming language compilers. They apply to the problem area of incomplete or erroneous documentation.

SDVS

SDVS is a collection of software tools with capabilities similar to those of other abstracted tools. Simulation is performed by AUT, FADEBUG-1, and TPL. Rollback resembles a function performed by AIDS. Analysis of these capabilities is found in evaluations of the associated software tools.

The SDVS post-run edit capability does not offer significant capabilities for microprogram development. The calling of analysis routines according to a user-supplied file is a minor convenience feature rather than a verification tool.

STRUCT

The single motivation for developing STRUCT was to enforce the restriction that programs must adhere to the concepts of structured programming. STRUCT essentially verifies that FORTRAN programs contain code which properly simulates well-structured constructs such as IF-THEN-ELSE, DO-WHILE, DO-UNTIL, etc. It was developed to compensate for the inadequate control constructs in FORTRAN. It is possible that with a properly designed high-level microprogramming language this capability would not be needed. However, unless the language explicitly prohibits the unrestricted GO TO, a tool such as STRUCT would still be useful in analyzing the source code for structure. By enforcing structured programming concepts, STRUCT aids in detecting violations of programming practices.

APPENDIX D
FUNCTIONAL APPLICABILITY OF TOOLS TO PROBLEM AREAS

Functional Applicability of Tools to Problem Areas

Problem Area	Tool Type	Tool Function With Respect to Problem Area
1. Incomplete or erroneous specifications	Formal verifiers	o Detect inconsistencies in specifications
	Requirements analysis aids	o Perform requirements analysis o Provide requirements tracing o Provide configuration control o Perform impact analysis
2. Violation of programming practices	Program auditors	o Detect error-prone constructs o Check interfaces for consistency o Check for use of nonstructured constructs o Check for coding standards o Check for uninitialized variables
3. Disagreement between code and detailed design specification	Formal verifiers	o Compare code with assertions or other specifications
	Execution monitors	o Provide execution trace o Examine memory and registers o Check for variable bounds violations
	Simulators/Symbolic executors	o Compare expected and actual outputs o Provide symbolic representation of program
	Flowchart generators	o Provide pictorial representation of program
	Program auditors	o Perform data flow analysis o Detect unused assignments o Detect uninitialized variables o Detect unreachable code o Detect infinite loops

Functional Applicability of Tools to Problem Areas (continued)

Problem Area	Tool Type	Tool Function With Respect to Problem Area
4. Incorrect accessing or storing of data	Execution monitors	o Provide line execution counts o Provide minimum/maximum values of variables o Provide snapshots of memory and registers
	Simulators/Symbolic executors	o Provide symbolic representation of outputs
	Formal verifiers	o Compare descriptions or assertions with code
	Program auditors	o Check interface consistency o Detect error-prone constructs o Check uninitialized variables o Note where variables are defined, redefined, undefined, and referenced
5. Errors in equation computation or arithmetic	Formal verifiers	o Prove consistency of assertions with code
	Simulators/Symbolic executors	o Compare equations with symbolic output
	Execution monitors	o Examine memory and registers during arithmetic operations o Provide line execution counts o Provide variable value bounds monitoring
6. Branching errors	Formal verifiers	o Provide assertions concerning the conditions for executing specific blocks of code

Functional Applicability of Tools to Problem Areas (continued)

Problem Area	Tool Type	Tool Function With Respect to Problem Area
	Execution monitors	o Provide execution location trace with snapshot dumps at branch points o Provide line execution counts showing which branches were taken
	Flowchart generators	o Produce flowcharts from code o Restructure code to conform to the single-entry, single-exit rules of well-structured programs
	Test case generators	o Generate test data for execution of the requested paths
	Simulators/Symbolic executors	o Provide automated path selection with case analysis and case description
7. Incorrect constants or data formats	Formal verifiers	o Assert properties of constants and data
	Simulators/Symbolic executors	o Examine storage locations of constants and data using snapshot dumps
8. Errors in timing and process allocation	Execution monitors	o Monitor real-time processing with breakpoints o Simulate timing of machine operations o Gather data on the timing of paths executed
	Program auditors	o Find maximum and minimum execution time for entire subroutines

Functional Applicability of Tools to Problem Areas (continued)

Problem Area	Tool Type	Tool Function With Respect to Problem Area
9. Errors in interruptibility and data coherency	Formal verifiers	o Permit interrupts to be asserted on or off
	Simulators/Symbolic executors	o Provide timing of machine operations, including interrupt handling
	Execution monitors	o Provide real-time tests of actual hardware and software, with breakpoints
10. Logic and sequence errors	Formal verifiers	o Assert sequence of operations and results, and compare assertions with code
	Flowchart generators	o Prepare flowcharts from code
	Simulators/Symbolic executors	o Detect logic and sequence errors based on expected vs. actual/symbolic outputs
	Execution monitors	o Use snapshot dumps to observe intermediate steps in execution counts o Provide cumulative line and branch execution counts
	Program auditors	o Detect unreferenced labels o Provide list of all statements that may follow each statement in execution o Detect unused assignments o Detect uninitialized variables
11. Erroneous use of system hardware/software/firmware	Simulators/Symbolic executors	o Run system software on the simulator o Compare actual outputs with expected results
	Execution monitors	o Monitor real-time execution of system hardware and software

Functional Applicability of Tools to Problem Areas (continued)

Problem Area	Tool Type	Tool Function With Respect to Problem Area
12. Incomplete or erroneous documentation	Formal verifiers	o Useful to the extent that code descriptions and assertions constitute documentation
	Cross-reference generators	o Generate cross-reference from code; augment or verify documentation
	Test case generators	o Verify user's manual or specifications by designing and executing test cases
	Flowchart generators	o Augment or verify documentation

APPENDIX E
MICROPROGRAMMING TOOL EVALUATION FORM

Microprogramming Tool Evaluation Form

Attribute	Guidelines for Assigning Value	Value
1. Range of application of the tool		
a. Language independence	1: Language independent 3: Handles more than one language 5: Handles a single lnguage	a ____
b. Nature of the limitations it imposes on subject microprograms (e.g., size, complexity)	1: None 2: Slight 3: Some 4: Considerable 5: Severe	b ____
c. Portability, i.e., number of machines on which it executes	1: Machine independent 2: Several common machines 3: One common machine 4: Several uncommon machines 5: One uncommon machine	c ____
- Weighted average of the values assigned to factors a, b, and c	$\dfrac{\sum_{i=1}^{3} W_i*(\text{value for ith factor})}{\sum_{i=1}^{3} W_i}$ where W_i is the weighting factor for the ith evaluation factor	____

Microprogramming Tool Evaluation Form (continued)

Attribute	Guidelines for Assigning Value	Value
2. Effectiveness		
a. Number of microprogram problem areas it addresses	1: 11-12 2: 8-10 3: 5-7 4: 2-4 5: 0-1	a ____
b. Usefulness of results (e.g., specificness, completeness)	1: Superior 2: Excellent 3: Good 4: Fair 5: Poor	b ____
- Weighted average of the values assigned to factors a and b	$\dfrac{\sum_{i=1}^{2} W_i * (\text{value for ith factor})}{\sum_{i=1}^{2} W_i}$ where W_i is the weighting factor for the ith evaluation factor	____

Microprogramming Tool Evaluation Form (continued)

Attribute	Guidelines for Assigning Value	Value
3. Evidence of reliability		
- Evidence that the tool has undergone comprehensive testing and has a history of producing accurate, consistent results	1: First-hand knowledge 2: Convincing evidence 3: Some evidence 4: Little evidence 5: No evidence	____
4. Usability		
a. Ease of the preparations required for a typical application	1: Very easy 2: Easy 3: Medium 4: Difficult 5: Very difficult	a ____
b. Ease of a typical application	1: Very easy 2: Easy 3: Medium 4: Difficult 5: Very difficult	b ____
c. Ease of interpreting the results	1: Very easy 2: Easy 3: Medium 4: Difficult 5: Very difficult	c ____

Microprogramming Tool Evaluation Form (continued)

Attribute	Guidelines for Assigning Value	Value
- Weighted average of the values assigned to factors a, b, and c	$\dfrac{\sum_{i=1}^{3} W_i * (\text{value for ith factor})}{\sum_{i=1}^{3} W_i}$ where W_i is the weighting factor for the ith evaluation factor	___
5. Initial cost		
For a tool or technique that is to be used as-is:	1: Under \$50K 2: \$50K to under \$100K 3: \$100K to under \$150K 4: \$150K to under \$200K 5: \$200K or more	___
o Cost to acquire \$________		
o Cost to install \$________		
o Cost of special equipment \$________		
o Cost of support software \$________		
Total \$________		
For a tool or technique that is to be modified from an existing tool or technique:		
o Cost to acquire existing tool or technique \$________		
o Cost to modify \$________		
o Cost of special equipment \$________		

Microprogramming Tool Evaluation Form (continued)

Attribute		Guidelines for Assigning Value	Value
o Cost of support software	$______		
Total	$______		
For a tool or technique that is to be newly developed:			
o Cost to develop	$______		
o Cost of special equipment	$______		
o Cost of support software	$______		
Total	$______		
6. Cost of application to a microprogram		1: Under $50 2: $50 to under $100 3: $100 to under $150 4: $150 to under $200 5: Over $200	___
o Computer cost	$______		
o Manpower cost	$______		
o Cost of special equipment usage	$______		
Total	$______		
o Product of number of applications expected for each microprogram times application cost	$______		

Microprogramming Tool Evaluation Form (continued)

Attribute	Guidelines for Assigning Value	Value
Weighted average of attributes 1 through 6	$\dfrac{\sum_{n=1}^{6} W_n*(\text{value for attribute } n)}{\sum_{n=1}^{6} W_n}$ where W_n is the weighting factor for attribute n	___

APPENDIX F
SUGGESTED PROGRAMMING PRACTICES

Microprogram development at RADC is performed for the QM-1 computer, using two languages:

- o MULTI, which resembles assembly language
- o SMITE, a higher-order language used for emulation

This appendix identifies programming practices which might be established for these languages and which could be monitored by an automated Programming Practices Monitor.

Language		
MULTI	SMITE	Programming Practice
		1. Program identification and labeling
X	X	o Use of descriptive comments at the beginning of the program
	X	o Use of descriptive comments at the beginning of each processor to describe inputs, outputs, interfaces with other processors, processor function, and error conditions returned or handled
X	X	o Use of descriptive comments at each branch point or code segment
X	X	o Adherence to naming conventions
X	X	o Initialing and dating of each line of code
		2. Parameter passing and routine interfaces
	X	o Use of the same number of arguments in calls as in the processor definition
	X	o Consistent placement of input parameters in relation to output parameters in the parameter list
	X	o Consistent declaration of parameters
	X	o Avoidance of constants and of arithmetic or logical expressions as arguments in calling statements
	X	o Use of a single return from each processor
		3. Status-flag setting
X	X	o Uniformity in status flags (e.g., 0 always means false)
X	X	o Uniformity in the method of setting status flags

Language		
MULTI	SMITE	Programming Practice
		4. Register usage
X		o If a register is equivalenced to a symbolic name, use of the name in all references to the register
X		o Reserving particular registers for particular functions
		5. Other practices
	X	o Consistent use of timing units
X	X	o Keeping modules within an established module length
	X	o Using indentation to delineate levels of declaration and nesting of blocks
	X	o Placing the DEFAULT declaration, if used, as the first data item declared in the main processor
		6. Instruction and data handling practices to be avoided
X		o Modification of instructions
X		o Use of data for instructions and vice versa
X	X	o Referencing or modifying restricted data
X	X	o Use of restricted instructions
X	X	o Use of specific error-prone instruction sequences
	X	o Use of a single storage location for multiple functions
	X	o Implicit conversions
X	X	o Use of embedded physical constants (e.g., 3.14159 for pi)
		7. Branching practices to be avoided
	X	o Branching outside a module
	X	o Backward branching
X	X	o Branching into a loop
	X	o Branching into a block
		8. Variable and label usage practices to be avoided
X	X	o Setting a variable but not using it
X	X	o Using a variable but not setting it
X	X	o Defining a variable or label but not referencing it

AD-A073 454 LOGICON INC SAN PEDRO CALIF F/G 9/2

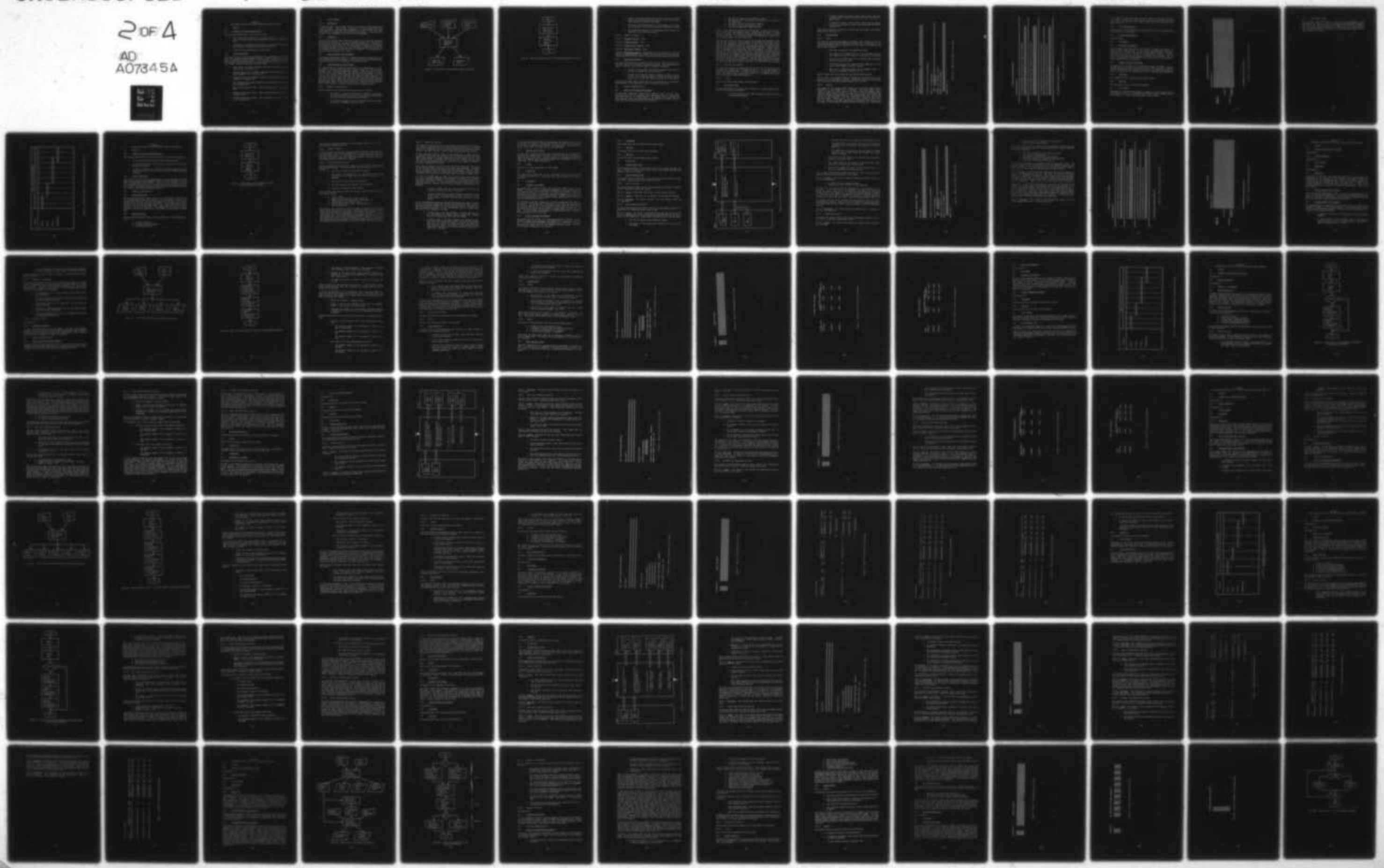
RELIABLE MICROPROGRAMMING.(U)

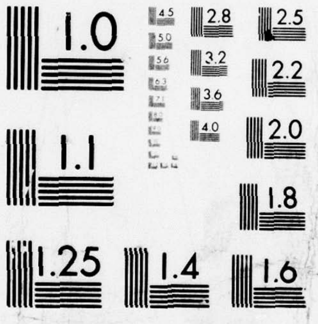
JUL 79 J RADATZ, J LAUB, G HAMACHI F30602-78-C-0079

UNCLASSIFIED DS-R79001 RADC-TR-79-173 NL

2 OF 4

AD A073454





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX G
FUNCTIONAL DESCRIPTION FOR THE MICROCODE COMPARISON PROGRAM

G1. GENERAL

G1.1 Purpose of Functional Description

This Functional Description is written to provide:

- o The system requirements to be satisfied, which will serve as a basis for mutual understanding between the user and the developer
- o Information on performance requirements, preliminary design, and user impacts, including fixed and continuing costs
- o A basis for the development of system tests

G1.2 Project References

This system is one of the microprogramming tools recommended by Logicon under the Reliable Microprogramming Contract, F30602-78-C-0079, with the Rome Air Development Center (RADC). These tools are intended to be used for microprogram verification in the System Architecture Evaluation Facility at RADC. Applicable documents are as follows:

- o RADC, PR No. B-7-3280, "Statement of Work for Reliable Microprogramming," 19 April 1977
- o Logicon, Report No. DS-R78026, "Reliable Microprogramming Program Management Plan," 29 March 1978
- o Logicon, Report No. DS-R78055, "Reliable Microprogramming Interim Report," 1 August 1978
- o DoD Automated Data System Documentation Standards Manual 4120.17-M, December 1972
- o RADC Report, RADC-TR-77-364, "SMITE Training Manual," 12 August 1977
- o Nanodata Corporation Report, "MULTI Micromachine Description," 2 February 1976
- o Nanodata Corporation Report, "QM-1 Hardware Level User's Manual," 31 March 1975

G2. SYSTEM SUMMARY

G2.1 Background

In February 1978, Logicon began work on the Reliable Microprogamming Contract for RADC. This project consisted of a study analyzing current microprogramming technology and recommending a set of tools that would contribute to microprogram verification and validation. The system described in this document is one of the tools recommended.

G2.2 Objectives

The purpose of the Microcode Comparison Program is to provide an automated means of identifying all of the differences between two versions of a microprogram. During the development and testing process, a microprogram may be modified many times. It is often useful for configuration control or debugging purposes to identify the differences between two versions or to establish that two versions are in fact identical. The Microcode Comparison Program automates this process.

G2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of a Microprogram Comparison Program. Identification of the differences between two versions of a microprogram is a manual process.

G2.4 Proposed Methods and Procedures

The proposed method of comparing two versions of a microprogram is to submit them to the Microcode Comparison Program along with a set of user-prepared inputs specifying comparison options. These options will identify the title of the comparison, the starting point and number of records (i.e., lines of code) to be compared, the record positions that are to be compared, the record types that are to be ignored in the comparison, if any, and the type of comparison report desired. The program will process and report on these inputs, then perform the comparison and produce the desired report based on the options specified. Figure G1 depicts the data flow for the program. Figure G2 shows the major processing steps.

G2.4.1 Summary of Improvements

Specific benefits to be gained from the program are as follows:

- o The need to identify differences by tedious, error-prone manual inspection of the code is eliminated, saving programmer time and providing more accurate results.
- o All project personnel can be provided with an accurate accounting of changes as they are made.

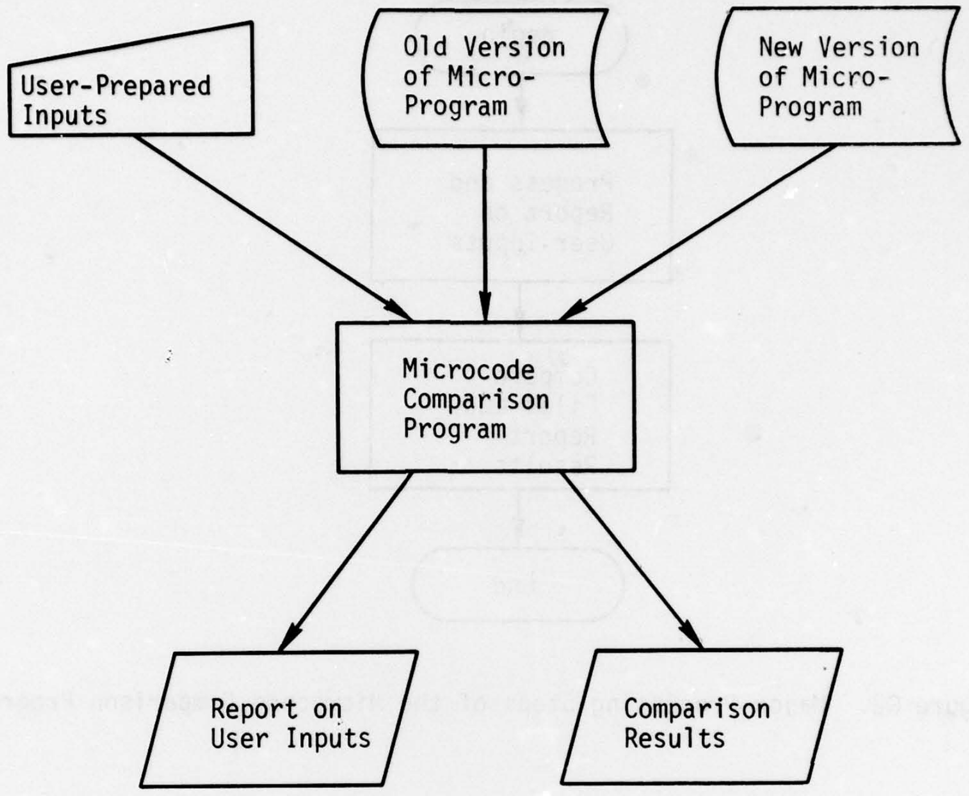

User-Prepared Inputs
Old Version of Micro-Program
New Version of Micro-Program
Microcode Comparison Program
Report on User Inputs
Comparison Results

Figure G1. Data Flow for the Microcode Comparison Program

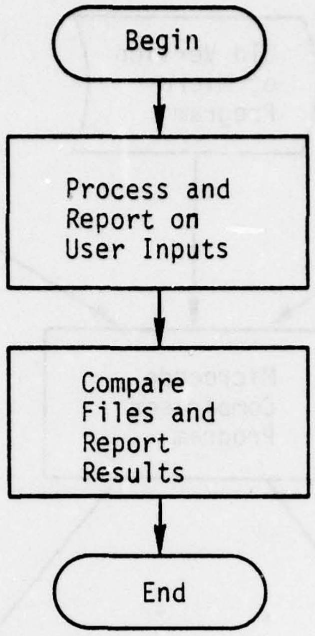

Begin
Process and Report on User Inputs
Compare Files and Report Results
End

Figure G2. Major Processing Steps of the Microcode Comparison Program

- Errors in a current version that did not exist in an earlier version can be pinpointed more easily if the exact differences between the versions are known.

- The final, certified version of a microprogram can be shown to be identical to the version that was successfully tested.

- The operational version of a microprogram can be shown to be identical to the certified version.

G2.4.2 Summary of Impacts

G2.4.2.1 <u>Equipment Impacts</u>. None.

G2.4.2.2 <u>Software Impacts</u>. None.

G2.4.2.3 <u>Organizational Impacts</u>. None.

G2.4.2.4 <u>Operational Impacts</u>. None.

G2.4.2.5 <u>Development Impacts</u>. Development of the system will require access by remote terminal to the RADC DEC System 20 for 3 hours per day during system design and 6 hours per day during system coding and testing.

G2.5 <u>Expected Limitations</u>

To limit execution time, an arbitrary limit will be placed on the size of the modifications the program is designed to handle. The comparison will cease if either of the following types of changes is detected:

- A block of more than 100 records deleted from the old version or inserted into the new version

- A block of more than 10 records inserted in place of a block of more than 10 deleted records, unless the blocks are of equal size, in which case they may contain up to 100 records

Changes that exceed these limitations will be referred to as "excessive modifications" in the remainder of this Functional Description.

G3. DETAILED CHARACTERISTICS

G3.1 <u>Specific Performance Requirements</u>

The Microcode Comparison Program shall accept as input a set of user-prepared inputs and two sequential files, designated the "old" version and the "new" version of a microprogram. The program shall interpret the user-prepared inputs and record the options specified in them. The options that may be specified shall be as follows:

- o The title to appear on the comparison report
- o The starting point and number of records to be included in the comparison
- o The portion of each record to be compared
- o The record types to be ignored, if any
- o The type of comparison report desired

Default values shall be provided for each option. A report shall be output, listing the user-prepared inputs, identifying any errors in their format or content, and identifying the options that are to be in effect for the comparison. If errors are present, the program shall request corrections and process the resulting inputs.

When the user inputs are free of errors, the program shall proceed to compare the two sequential files according to the options specified. Beginning at the designated starting point of each file, the program shall compare the specified portions of corresponding records in the two files, ignoring any records of the types specified by the user. When a difference is detected, the program shall determine whether it results from a deletion of records from the old version, an insertion of records into the new version, or a combination of the two. The results of the comparison shall be output in a report of the type designated by the user. Premature termination of the program shall occur if an excessive modification, as defined in Section G2.5, is detected. Otherwise the program shall process the number of records specified by the user and shall terminate normally. An output message shall identify the type of termination that occurred.

G3.1.1 Accuracy and Validity

If there are no excessive modifications in the portions of the files being compared, the program shall accurately report all of the differences in these portions of the files. If there are one or more excessive modifications, the program shall accurately report all differences that occur before the first of these modifications.

G3.1.2 Timing

There are no timing requirements on the program.

G3.2 System Functions

The Microcode Comparison Program shall consist of a single program incorporating the following functions:

- o A Control function, which shall invoke the other functions in the proper sequence

- An Option Recording function, which shall process the user-prepared inputs, record the specified options, and report on these inputs.

- A Comparison function, which shall compare the two sequential files and prepare a report identifying the differences between them

These three functions suffice to satisfy the performance requirements identified in Section G3.1.

G3.3 Inputs/Outputs

G3.3.1 Inputs

The inputs to the Microcode Comparison Program shall consist of a set of user-prepared option specifications and the two files to be compared. The option specifications may be submitted via card reader or terminal and may include the following:

- The title to appear on the comparison report

- The range of the comparison, i.e., the starting point for the comparison and the number of records to be compared

- The portion of each record to be compared (any contiguous subset of positions 1-132)

- The record types to be ignored in the comprison (for example, blank comments used for spacing)

- The type of comparison report to be produced (Type 1 or Type 2, as described in Section G3.3.2)

Default values shall be provided for any omitted specifications.

The two files to be compared shall be sequential files with logical records up to 132 characters in length. The records may be of any format. The files may be on punched cards, magnetic tape, or disk.

G3.3.2 Outputs

The output of the program shall consist of two reports, which may be directed to a line printer or to a terminal. The first report, illustrated in Figure G3, shall contain a listing of the user-prepared inputs with any errors identified and, if there are no errors, a listing of the options in effect for the comparison. The second report shall contain the results of the comparison. The user shall have a choice of two formats for this report. Report Type 1, as illustrated in Figure G4, shall consist of a series of messages identifying all deletions and insertions in the order in which they are detected during the comparison. Report Type

```
MICROCODE COMPARISON PROGRAM

USER INPUTS

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


OPTIONS IN EFFECT FOR THE COMPARISON

RANGE OF THE COMPARISON:

   FROM:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   TO:    XXXXXXXXXXXX

COLUMNS TO BE COMPARED: XXX-XXX

RECORD TYPES TO BE IGNORED:

   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

REPORT TYPE:  X
```

Figure G3. Format of the Report on User Inputs

```
                    MICROCODE COMPARISON PROGRAM

          COMPARISON RESULTS
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN DELETED FROM THE OLD VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN INSERTED INTO THE NEW VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN DELETED FROM THE OLD VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          AND THE FOLLOWING RECORDS HAVE BEEN INSERTED IN THEIR PLACE

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************


          THE COMPARISON HAS BEEN COMPLETED.
```

Figure G4. Format of Comparison Report Type 1

2, as shown in Figure G5, shall list each record in the portion of the files that underwent comparison and shall identify the records that have been deleted or inserted. Both report types shall begin with the user-specified title.

G3.4 Data Characteristics

One or both of the microprogram versions used as input to the program may be contained in a data base.

G3.5 Failure Contingencies

Not applicable.

G4. ENVIRONMENT

G4.1 Equipment Environment

The Microcode Comparison Program will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 25K words of main memory to operate. The program will be stored on disk. The user-prepared inputs will be entered via card reader or terminal. The two files to be compared may be stored on punched cards, magnetic tape, or disk. The output will be directed to a line printer or to a terminal. No new equipment will be required.

G4.2 Support Software Environment

The system will be written in a structured version of FORTRAN. The required support software will consist of an RADC-supplied structured FORTRAN preprocessor and a FORTRAN IV compiler. The operating system for which the system will be developed will be the TOPS20AN Operating System.

G4.3 Interfaces

The program will not interface with any other system.

G4.4 Security

The program will have no classified components.

G5. COST FACTORS

Development of the Microcode Comparison Program will require approximately 3 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the program.

```
                             MICROCODE COMPARISON PROGRAM

                   COMPARISON RESULTS
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                      .
                      .
                      .
                      .

                   THE COMPARISON HAS BEEN COMPLETED.
```

Figure G5. Format of Comparison Report Type 2

G6. DEVELOPMENTAL PLAN

The Microcode Comparison Program is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Microcode Comparison Program is given in Figure G6.

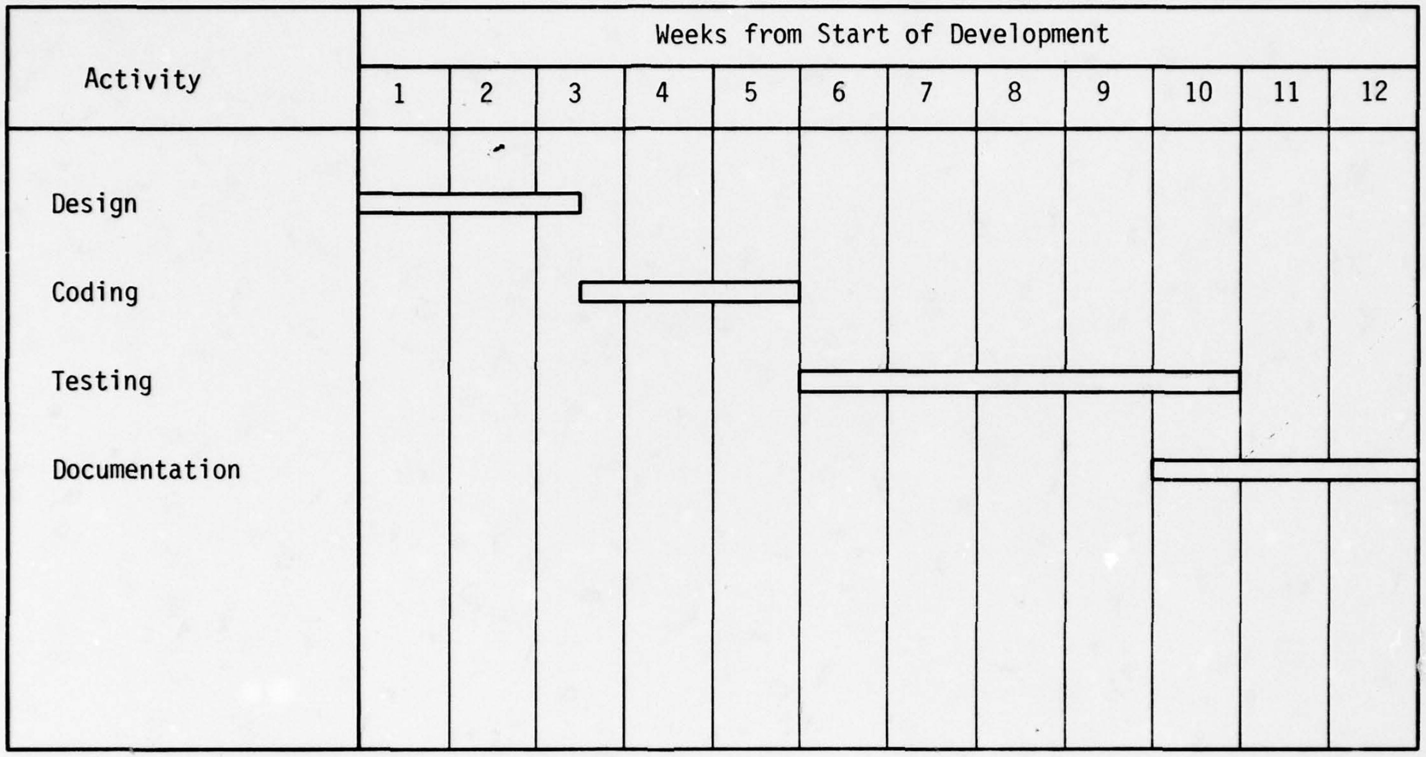


Figure G6. Development Schedule for the Microcode Comparison Program

APPENDIX H
SYSTEM SPECIFICATION FOR THE MICROCODE COMPARISON PROGRAM

H1. GENERAL

H1.1 Purpose of the System Specification

This System Specification is written to fulfill the following objectives:

- o To provide detailed definition of the system functions
- o To communicate details of the on-going analysis to the user's operational personnel
- o To define in detail the interfaces with other systems and subsystems and the facilities to be utilized for accomplishing the interfaces

H1.2 Project References

This system is one of the microprogramming tools recommended by Logicon under the Reliable Microprogramming Contract, F30602-78-C-0079, with the Rome Air Development Center (RADC). These tools are intended to be used for microprogram verification in the System Architecture Evaluation Facility at RADC. Applicable documents include the Functional Description for this tool and the documents referenced therein.

H.2 SUMMARY OF REQUIREMENTS

H2.1 System Description

The Microcode Comparison Program shall be a tool designed to identify all of the differences between two versions of a microprogram. Inputs to the program shall consist of the two microprogram versions and a set of user-prepared inputs specifying comparison options. The program shall process and report on the user-prepared inputs, then perform the comparison according to the options specified. The system shall consist of a single program. Figure H1 illustrates the relationship of the program's components.

H2.2 System Functions

The Microcode Comparison Program shall be comprised of the following functions:

- o A Control function
- o An Option Recording function
- o A Comparison function

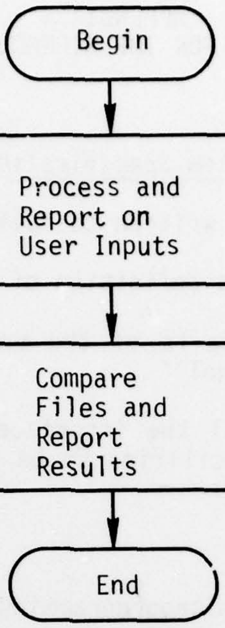

Begin
Process and Report on User Inputs
Compare Files and Report Results
End

Figure H1. Relationship of the Components of the Microcode Comparison Program

The following paragraphs identify the requirements that are to be satisfied by each of these functions.

H2.2.1 Control Function

The Control function shall be responsible for invoking the other functions of the program in the proper sequence and for terminating the program at the appropriate time. It shall begin by invoking the Option Recording function, then shall invoke the Comparison function, then shall terminate the program when this function is complete.

H2.2.2 Option Recording Function

The Option Recording function shall accept as input a set of user-prepared option specifications. The options that may be specified shall be as follows:

- o The title to appear on the comparison report
- o The range of the comparison, i.e., the record on which the comparison is to begin and the number of records to be compared
- o The portion of each record to be compared
- o The record types to be ignored in the comparison
- o The type of comparison report desired

Each of these specifications shall be optional; default options shall be provided as follows:

- o Title: Blank
- o Range of the comparison: Beginning to end
- o Record portions to be compared: Characters 1-72
- o Record types to be ignored: None
- o Desired report type: Type 1 (as described in Section H4.2.3.2)

The function shall check the inputs for errors in format and content and shall record the options to be used in the comparison, providing default values for any options not specified. The function shall prepare a report listing each user input and identifying any errors detected. If errors are present, the function shall request corrections, then process and list the corrected inputs as it did the original ones. When all of the inputs have been successfully processed, the function shall conclude the report by identifying the options that are to be in effect for the comparison.

H2.2.3 Comparison Function

The Comparison function shall read and compare the records of two sequential files, designated the "old" version and the "new" version. It shall perform this comparison using the options recorded by the Option Recording function, and shall prepare a report presenting the comparison results.

If the specified starting point for the comparison is other than the beginning of the files, the function shall search each file for the first record that matches the record image given as the starting point. Once the starting point is reached, the function shall compare the specified portions of corresponding records in the two files, ignoring records of the types designated by the user.

When a pair of records is found to match during the comparison, the action of the function shall depend upon the report type selected. If Report Type 1 has been selected, no output shall be generated; if Report Type 2 has been selected, a copy of the matching pair of records shall be output. A maximum of 120 characters of the record shall be output in Report Type 2.

If a pair of records does not match, the function shall search ahead in the files to determine whether the difference is the result of a deletion of records from the old version, an insertion into the new version, or a combination of the two. To limit execution time, the function shall terminate the search if either of the following "excessive modifications" is detected:

- o A block of more than 100 records deleted from the old version or inserted into the new version
- o A block of more than 10 records inserted in place of a block of more than 10 deleted records, unless the blocks are of equal length, in which case they may contain up to 100 records

If the correspondence cannot be reestablished because the number of deletions or insertions exceeds this limitation, the function shall output a termination message identifying the last matching records and shall perform no further processing. If the correspondence can be reestablished, the function shall proceed as follows:

- o If Report Type 1 has been selected, a message shall be output identifying the deleted records and/or the inserted records; the comparison shall then be resumed.
- o If Report Type 2 has been selected, any deleted records shall be listed and labeled as such, then any inserted records shall be listed and labeled as such; the comparison shall then be resumed. A maximum of 120 characters of each record shall be output to allow space for the indication of deletion or insertion.

If no excessive modifications are encountered, the comparison shall continue until the number of records specified by the user has been compared or the ends of the files are reached. In either case, a message shall be output stating that the comparison has been completed.

H2.3 Accuracy and Validity

If there are no excessive modifications in the portions of the files being compared, the program shall accurately report all of the differences in these portions of the files. If there are one or more excessive modifications, the program shall accurately report all differences that occur before the first of these modifications.

H2.4 Timing

There are no timing requirements on the program.

H2.5 Flexibility

The program shall be written in such a way that the limit on the number of consecutive deletions or insertions that can be handled can be easily modified.

H3. ENVIRONMENT

H3.1 Equipment Environment

The system shall operate on RADC's DEC System 20, which is tied to the ARPANET. The System 20 currently has 256K words of main memory, two 44-million-word disks; four 9-track 800- or 1600-bpi tape drives; a 300-card-per-minute card reader; a 132-column line printer capable of printing 231 lines per minute; and 16 asynchronous channels. Changes planned for early 1979 include expansion of main memory to 512K words and the addition of two more 44-million-word disks.

The Microcode Comparison Program will require approximately 25K words of main memory. The program shall be stored on disk, requiring approximately 15K words of storage for the source code, object code, and load module. The user-prepared inputs may be entered via card reader or terminal. The two files to be compared may be stored on punched cards, magnetic tape, or disk. The output shall be directed to a line printer or to a terminal. No new equipment will be required.

H3.2 Support Software Environment

The system shall be written in a structured version of FORTRAN. Support software required for development and maintenance of the program will consist of an RADC-supplied FORTRAN preprocessor and a FORTRAN IV compiler. The operating system shall be the TOPS20AN Operating System. No additional support software will be required.

H3.3 Interfaces

The program shall not interface with any other system.

H3.4 Security

The program shall have no classified components.

H3.5 Controls

No controls shall be established by the program.

H4. DESIGN DATA

H4.1 System Logical Flow

The Microcode Comparison Program shall consist of a single computer program. Figure H2 indicates the logical flow of the program and shows the program's inputs and outputs.

H4.2 Function Descriptions

The program shall be comprised of the three functions identified in Section H2.2. Design data for each function follow.

H4.2.1 Control Function

The Control function shall fulfill the requirements specified in Section H2.2.1. Additional design data follow.

H4.2.1.1 Inputs: There shall be no inputs to the Control function.

H4.2.1.2 Outputs: There shall be no outputs from the Control function.

H4.2.1.3 Data Base: The Control function shall not create or make use of a data base.

H4.2.2 Option Recording Function

The Option Recording function shall fulfill the requirements specified in Section H2.2.2. Additional design data follow.

H4.2.2.1 Inputs: The inputs to the Option Recording function shall consist of user-prepared inputs specifying the options that are to be in effect for the comparison. These inputs may specify any of the following:

- o The title to appear on the comparison report
- o The range of the comparison, which may be any of the following:

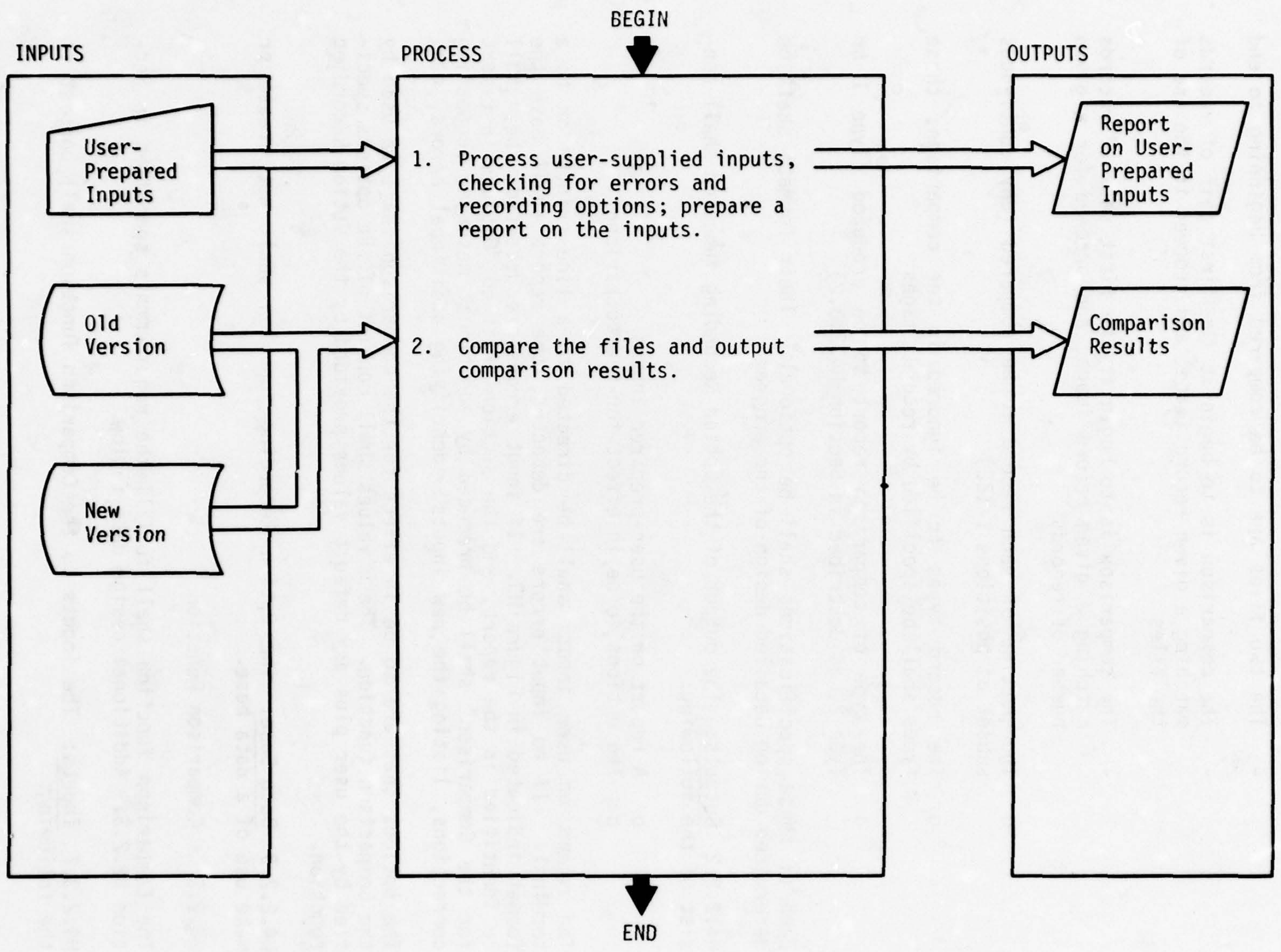

BEGIN
INPUTS
PROCESS
OUTPUTS
User-Prepared Inputs
Old Version
New Version
1. Process user-supplied inputs, checking for errors and recording options; prepare a report on the inputs.
2. Compare the files and output comparison results.
Report on User-Prepared Inputs
Comparison Results
END

Figure H2. System Logical Flow for the Microcode Comparison Program

- The two files are to be compared from beginning to end
- The comparison is to begin at the first pair of records matching a given record image, and proceed to the end of the files
- The comparison is to begin at the first pair of records matching a given record image, and proceed for a given number of records

o The portion of each record to be compared (any contiguous subset of positions 1-132)

o The record types to be ignored in the comparison; these types shall be specified by record images

o The type of comparison report to be produced (Type 1 or Type 2, as described in Section H4.2.3.2)

Each of these specifications shall be optional. Their formats shall be determined during detailed design of the program.

H4.2.2.2 Outputs: The output of the Option Recording function shall consist of the following:

o A report on the user-prepared inputs
o The options to be in effect for the comparison

The report on user inputs shall be directed to a line printer or to a terminal. If no input errors are detected, the report shall have the format indicated in Figure H3. If input errors are detected, they shall be identified in the report, and the section entitled "Options in Effect for the Comparison" shall be preceded by additional messages requesting corrections, listing the new inputs, identifying additional errors, etc.

The options that are to be in effect for the comparison shall be used by the Comparison function. These values shall consist of the options specified by the user plus any default values provided by the Option Recording function.

H4.2.2.3 Data Base: The Option Recording function shall not create or make use of a data base.

H4.2.3 Comparison Function

The Comparison function shall fulfill the requirements specified in Section H2.2.3. Additional design data follow.

H4.2.3.1 Inputs: The inputs to the Comparison function shall consist of the following:

```
MICROCODE COMPARISON PROGRAM

USER INPUTS

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


OPTIONS IN EFFECT FOR THE COMPARISON

RANGE OF THE COMPARISON:

   FROM:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   TO:    XXXXXXXXXXXX

COLUMNS TO BE COMPARED: XXX-XXX

RECORD TYPES TO BE IGNORED:

   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

REPORT TYPE:  X
```

Figure H3. Format of the Report on User Inputs

- o The options to be in effect for the comparison
- o The two files to be compared

The options that are to be in effect for the comparison shall be recorded by the Option Recording function. These options shall include values for each of the following:

- o The title to appear on the comparison report
- o The range of the comparison
- o The portion of each record to be compared
- o The record types to be ignored in the comparison
- o The type of comparison report desired

The two files that are to be compared shall be sequential files. The logical records in each file may be up to 132 characters in length and may be of any format. The files may be input from punched cards, magnetic tape, or disk. There shall be no restriction on their volume.

H4.2.3.2 Outputs: The output of the Comparison function shall consist of a report of the comparison results. The title and format of the report shall be as specified in the options. Report Type 1 shall have the format indicated in Figure H4. It shall identify, through a series of messages, all deletions and insertions in the order in which they are detected during the comparison. Report Type 2 shall have the format indicated in

Figure H5. It shall list each record in the portion of the files being compared and shall identify the records that have been deleted or inserted. The maximum number of characters of each record that may be output in Report Type 2 shall be 120, in order to allow space for the indication of deletion or insertion. Both report types shall indicate whether the comparison was successfully completed or whether an excessive modification caused premature termination of the program. In the latter case, the last pair of matching records shall be identified.

H4.2.3.3 Data Base: One or both of the microprogram versions that are being compared may reside in a system data base.

```
                    MICROCODE COMPARISON PROGRAM

          COMPARISON RESULTS
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN DELETED FROM THE OLD VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN INSERTED INTO THE NEW VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************
          THE FOLLOWING RECORDS HAVE BEEN DELETED FROM THE OLD VERSION

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

          AND THE FOLLOWING RECORDS HAVE BEEN INSERTED IN THEIR PLACE

          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
****************************************************************************************************


          THE COMPARISON HAS BEEN COMPLETED.
```

Figure H4. Format of Comparison Report Type 1

```
                             MICROCODE COMPARISON PROGRAM

                   COMPARISON RESULTS

                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
***DEL***          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         ***INS*** XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                      .
                      .
                      .
                      .

                   THE COMPARISON HAS BEEN COMPLETED.
```

Figure H5. Format of Comparison Report Type 2

APPENDIX I
FUNCTIONAL DESCRIPTION FOR THE MULTI CROSS-REFERENCE GENERATOR

I1. GENERAL

I1.1 Purpose of Functional Description

See Appendix G.

I1.2 Project References

See Appendix G.

I2. SYSTEM SUMMARY

I2.1 Background

See Appendix G.

I2.2 Objectives

The purpose of the MULTI Cross-Reference Generator is to provide cross-references for the variables (i.e., registers and symbolic names) and labels used in a microprogram written in MULTI. The cross-references will provide descriptive information about each entry and will identify all program statements in which the entry is defined, set, or used. These cross-references may be used for microprogram development, testing, and maintenance.

I2.3 Existing Methods and Procedures

MULTI microprogram development at RADC is currently performed without the aid of a Cross-Reference Generator. To determine the characteristics and usage of a variable or label beyond what is provided by the MULTI assembler, the user must inspect the source code manually.

I2.4 Proposed Methods and Procedures

The proposed method for identifying the characteristics and usage of each variable and label in a MULTI microprogram is to submit the source code to the MULTI Cross-Reference Generator. Based on user-selected options, the program will provide the following:

- o A complete listing of the microprogram, with each statement numbered
- o A cross-reference of all variables used in the program, describing each variable and identifying the statements in which it is declared, set, or used

- A cross-reference of all labels used in the program, identifying the statements in which each is defined or referenced

Figure I1 depicts the data flow for the system. Figure I2 shows its major processing steps.

I2.4.1 Summary of Improvements

During the development and testing of a MULTI microprogram, it is often useful to determine exactly how and where a particular variable or label has been used. The MULTI Cross-Reference Generator will provide this information in a concise, accurate manner. Specific benefits to be gained from use of the program are as follows:

- The information in the cross-references can be used in program debugging.
- The cross-references can be used to determine the effects of proposed changes to the code.
- Unreferenced labels can be identified, as can unset and unused variables.
- Adherence to naming conventions can be verified by inspection of the cross-references.
- The cross-references can serve as a permanent part of the microprogram documentation.

I2.4.2 Summary of Impacts

See Appendix G.

I2.5 Expected Limitations

To limit execution time and core requirements, the MULTI Cross-Reference Generator will be designed to handle a MULTI microprogram that is syntactically correct and that has no more than 5,000 labels, 2,000 variables, and 25,000 total references to labels and variables.

I3. DETAILED CHARACTERISTICS

I3.1 Specific Performance Requirements

The MULTI Cross-Reference Generator shall accept as input a set of user-prepared inputs and the source code of a microprogram written in MULTI. The user-prepared inputs shall specify the following options:

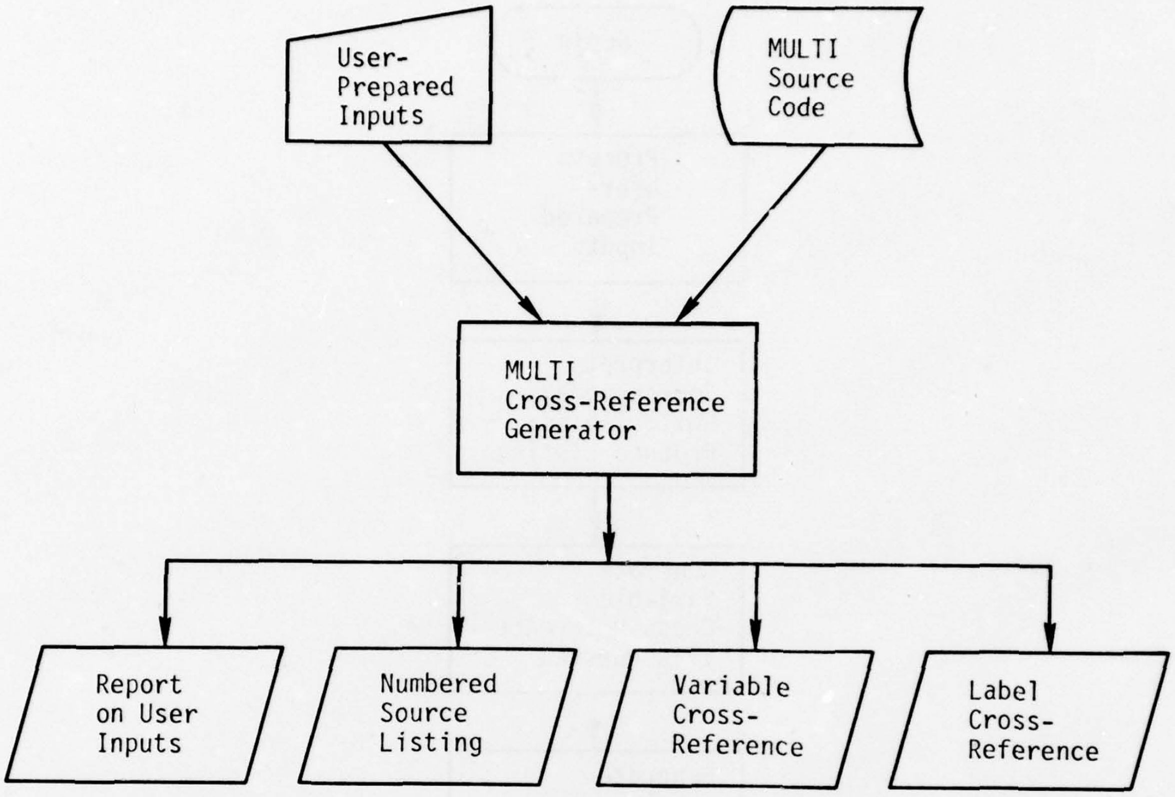


Figure I1. Data Flow for the MULTI Cross-Reference Generator

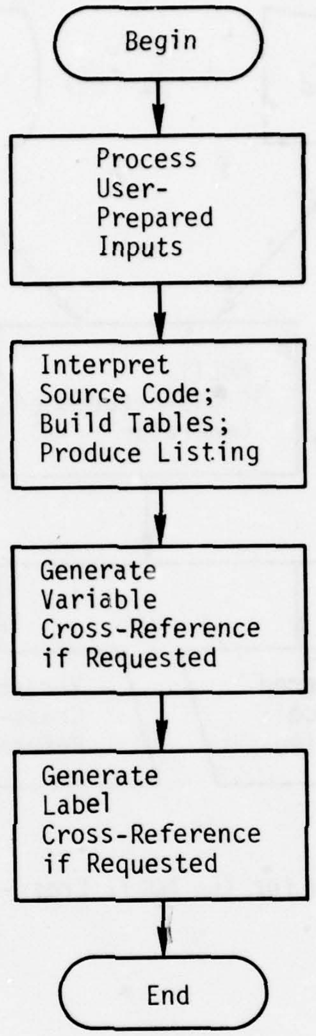

Begin
Process User-Prepared Inputs
Interpret Source Code; Build Tables; Produce Listing
Generate Variable Cross-Reference if Requested
Generate Label Cross-Reference if Requested
End

Figure I2. Major Processing Steps in the MULTI Cross-Reference Generator

- o The types of cross-references to be generated (variable cross-reference and/or label cross-reference)
- o Whether or not the program should generate these cross-references if problems are encountered in processing the MULTI source code
- o The number of columns to appear in each line of output (80 or 132)

Default values shall be provided for each option. A report shall be prepared identifying each input and indicating the options that will be in effect for the program.

If errors are detected in the user-prepared inputs, the system shall request corrections and process the resulting inputs. When the inputs are free of errors, the program shall process each source statement in the MULTI microprogram as follows:

- o Assign the statement a sequence number
- o Output a copy of the statement, along with its sequence number, in a listing of the microprogram
- o Interpret the contents of the statement and store relevant information in tables to be used for generating the cross-references

Relevant information to be stored in the tables shall include the following:

- o The name of each variable used in the microprogram, along with:
  - - The sequence number of the statement in which it is explicitly defined
  - - The sequence numbers of all statements in which it is set
  - - The sequence numbers of all statements in which it is used
- o Each label used in the microprogram, along with:
  - - The sequence number of the statement to which it is assigned
  - - The sequence numbers of all statements in which it is referenced

If a source statement cannot be interpreted during processing of the microprogram, a message indicating this problem shall accompany the statement in the numbered listing and the system shall proceed to the next source statement. If any of the numerical limitations given in Section I2.5 is exceeded during the processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

When all source statements have been processed, the microprogram shall proceed as follows:

- o If no problems were encountered during source code processing, the system shall proceed to generate the cross-references requested by the user.
- o If problems were encountered, the system shall determine from the user inputs whether to generate the requested cross-references despite the problems.

If the cross-references are not to be generated, an explanatory message shall be output and execution shall terminate. If they are to be generated, the system shall alphabetize the entries in each selected cross-reference and shall output the cross-references, incorporating the information stored in the tables during the processing of the source code and using the report width specified by the user.

I3.1.1 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

I3.1.2 Timing

There are no timing requirements on the system.

I3.2 <u>System Functions</u>

The MULTI Cross-Reference Generator shall consist of a single program incorporating the following functions:

- o A Control function, which shall invoke the other functions in the proper sequence
- o A User Input Processing function, which shall process the user-prepared inputs
- o A Source Code Processing function, which shall interpret each source statement, assign it a sequence number, produce a source listing, and build the tables needed for cross-reference generation

- A Variable Cross-Reference function, which shall generate the variable cross-reference

- A Label Cross-Reference function, which shall generate the label cross-reference

These five functions suffice to satisfy the performance requirements identified in Section I3.1.

I3.3 Inputs/Outputs

I3.3.1 Inputs

The inputs to the MULTI Cross-Reference Generator shall consist of user-prepared inputs and the source code of a microprogram written in MULTI. The user-prepared inputs shall be as follows:

- Specification of the types of cross-references desired: variable cross-reference and/or label cross-reference

- Specification of whether or not to generate the selected cross-references if problems are encountered during processing of the microprogram source code

- Specification of the number of columns to be used in each line of the system outputs: 80 or 132

These specifications may be input via a card reader or a terminal. The MULTI source code may be entered via punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image.

I3.3.2 Outputs

The output of the system shall consist of the following reports:

- A report on the user-prepared inputs
- A numbered listing of the microprogram source code
- A variable cross-reference, if requested
- A label cross-reference, if requested

The formats of these reports shall be as indicated in Figures I3, I4, I5, and I6, respectively. The reports shall be directed to a line printer or to a terminal.

I3.4 Data Characteristics

The MULTI microprogram to be processed may be contained in a system data base. In addition, the information on program variables and labels may be stored in a system data base until cross-reference generation is complete.

```
MULTI CROSS-REFERENCE GENERATOR

USER INPUTS
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT

SELECTED CROSS-REFERENCES
         XXXXXXXXXXXXXXXXX
         XXXXXXXXXXXXXXXXX

IN CASE OF SOURCE CODE PROBLEMS:  XXXXXXXXX
NUMBER OF COLUMNS TO BE USED:  XXX
```

Figure I3. Format of the Report on User Inputs

```
STATEMENT #            SOURCE STATEMENT

  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    .                    .
    .                    .
    .                    .
```

Figure I4. Format of the Numbered Source Listing

```
                         VARIABLE CROSS-REFERENCE

                                   *** STATEMENT NUMBERS ***
VARIABLE NAME                   DEFINED      SET      REFERENCED
  XXXXXXXX                       XXXXX      XXXXX       XXXXX
                                              .           .
                                              .           .
                                              .           .
                                              .           .
  XXXXXXXX                       XXXXX      XXXXX       XXXXX
                                              .           .
                                              .           .
                                              .           .
                                              .           .
  XXXXXXXX                       XXXXX      XXXXX       XXXXX
                                              .           .
                                              .           .
                                              .           .
                                              .           .

     .                              .
     .                              .
     .                              .
```

Figure I5. Format of the Variable Cross-Reference

```
                        LABEL CROSS-REFERENCE

                               *** STATEMENT NUMBERS ***
LABEL                        DEFINED          REFERENCED
XXXXXXXX                      XXXXX        XXXXX  XXXXX  XXXXX
                                             :      :      :
XXXXXXXX                      XXXXX        XXXXX  XXXXX  XXXXX
                                             :      :      :
XXXXXXXX                      XXXXX        XXXXX  XXXXX  XXXXX
                                             :      :      :
    .                            .
    .                            .
    .                            .
```

Figure I6. Format of the Label Cross-Reference

I3.5 Failure Contingencies

Not applicable.

I4. ENVIRONMENT

I4.1 Equipment Environment

The MULTI Cross-Reference Generator will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 60K words of main memory to operate. The program will be stored on disk. The user-prepared inputs may be entered via card reader or terminal. The MULTI source code may be stored on punched cards, magnetic tape, or disk. The output will be directed to a line printer or to a terminal. No new equipment will be required.

I4.2 Support Software Environment

See Appendix G.

I4.3 Interfaces

The system will not interface with any other system.

I4.4 Security

The system will have no classified components.

I5. COST FACTORS

Development of the MULTI Cross-Reference Generator will require approximately 6 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the system.

I6. DEVELOPMENTAL PLAN

The MULTI Cross-Reference Generator is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract.

RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the MULTI Cross-Reference Generator is given in Figure I7.

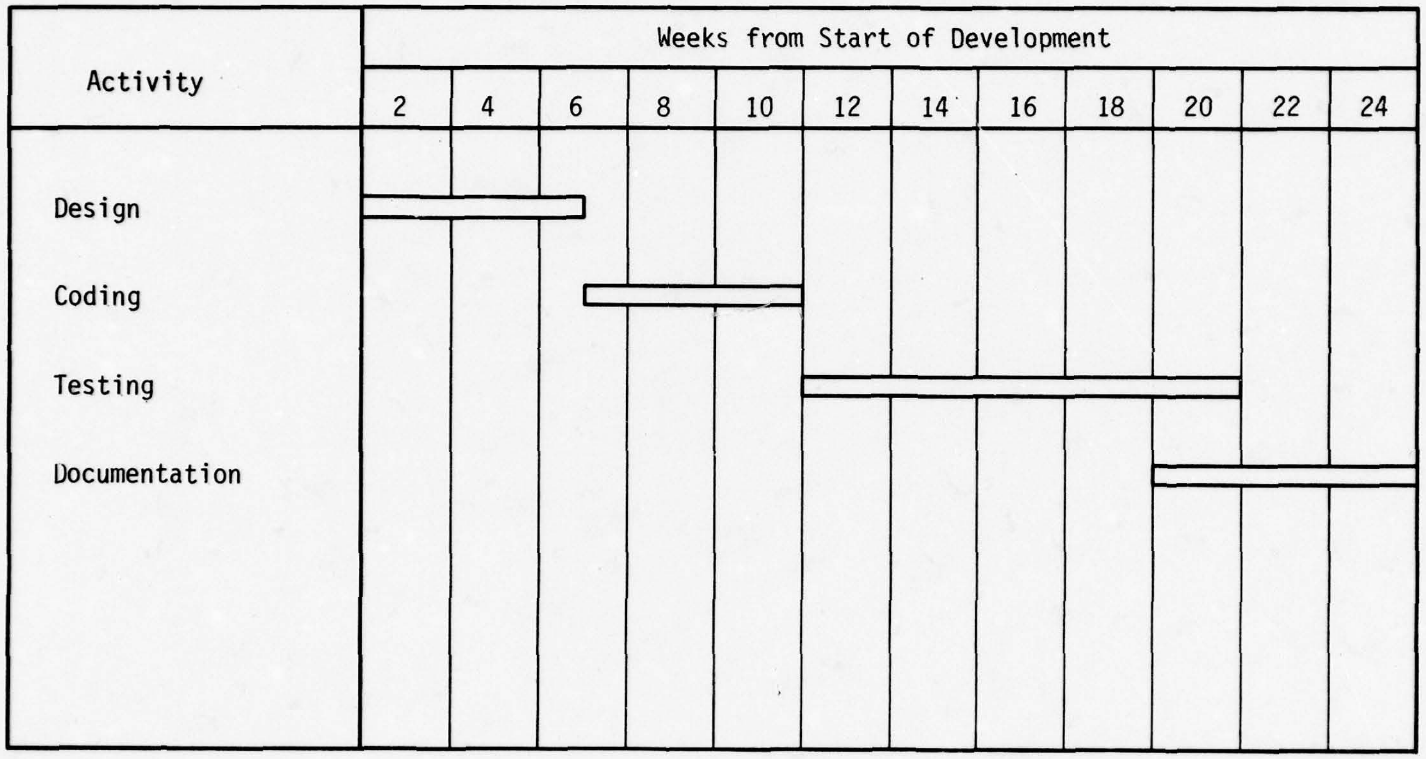


Figure I7. Development Schedule for the MULTI Cross-Reference Generator

APPENDIX J
SYSTEM SPECIFICATION FOR THE MULTI CROSS-REFERENCE GENERATOR

J1. GENERAL

J1.1 Purpose of the System Specification

See Appendix H.

J1.2 Project References

See Appendix H.

J2. SUMMARY OF REQUIREMENTS

J2.1 System Description

The MULTI Cross-Reference Generator shall provide cross-references for the registers, symbolic names, and labels in a microprogram written in MULTI. (Registers and symbolic names will hereafter be referred to as variables.) The cross-references shall provide descriptive information about each entry and shall identify all microprogram statements in which the entry is defined, set, or used. The system shall consist of a single program. Figure J1 illustrates the relationship of the program's components.

J2.2 System Functions

The MULTI Cross-Reference Generator shall be a single program incorporating the following functions:

- o A Control function
- o A User Input Processing function
- o A Source Code Processing function
- o A Variable Cross-Reference function
- o A Label Cross-Reference function

The following paragraphs identify the requirements to be satisfied by each of these functions.

J2.2.1 Control Function

The Control function shall invoke the other functions of the program in the proper sequence. It shall begin by invoking the User Input Processing function, which shall record the following values:

- o Cross-reference selection values, indicating which of the cross-references are to be generated (variable cross-reference and/or label cross-reference)

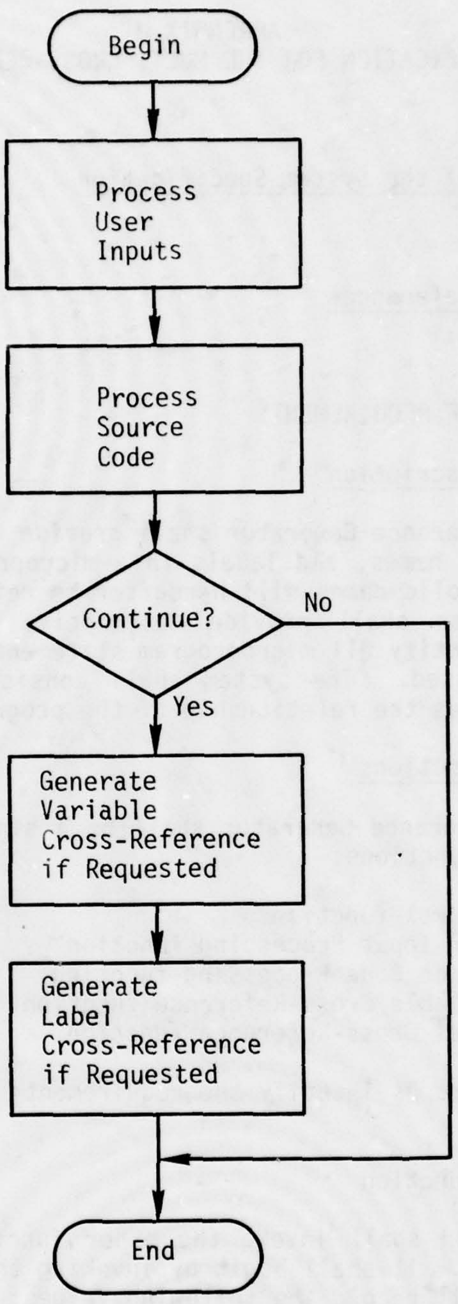

Begin
Process User Inputs
Process Source Code
Continue?
No
Yes
Generate Variable Cross-Reference if Requested
Generate Label Cross-Reference if Requested
End

Figure J1. Relationship of the Components of the MULTI Cross-Reference Generator

- A "continuation option," indicating whether these cross-references are to be generated if problems are encountered in processing the source code

The Control function shall next invoke the Source Code Processing function. This function will indicate whether problems were encountered in processing the source code. If problems were encountered and the continuation option indicates that the program should not continue, the Control function shall output an explanatory message and terminate the program. If no problems were encountered, or if problems were encountered but the continuation option indicates that the program should proceed, the Control function shall use the cross-reference selection values to determine which of the following functions to invoke:

- The Variable Cross-Reference function
- The Label Cross-Reference function

The appropriate functions shall be invoked, and when they have completed their processing the Control function shall terminate the program.

J2.2.2 User Input Processing Function

The User Input Processing function shall accept as input a set of user-prepared inputs specifying program options. The options that may be specified shall be as follows:

- The cross-references to be generated (variable cross-reference and/or label cross-reference)

- Whether the program should generate these cross-references if problems are encountered in processing the MULTI source code

- The number of columns to be used in each line of program output (80 or 132)

Each of these specifications shall be optional; default values shall be provided as follows:

- Cross-references to be generated: Both
- Program action if problems are encountered: Continue
- Number of columns: 132

The function shall check the inputs for errors in format and content and shall record the options specified, providing default values as required. A report shall be prepared listing each user input and identifying any errors detected. If errors are present, the function shall request and process corrections. When all of the inputs have been successfully processed, the function shall conclude the report by identifying the options that are to be in effect for the program.

J2.2.3 Source Code Processing Function

The Source Code Processing function shall accept as input the source code of a microprogram written in MULTI. The function shall process each source statement of the microprogram as follows:

- o Assign the statement a sequence number
- o Output a copy of the statement, along with its sequence number, in a listing of the program
- o Interpret the contents of the statement and store relevant information in tables to be used for generating the cross-references

Source code comments shall be copied into the program listing without being interpreted or assigned a sequence number.

The information that is to be stored in tables shall be as follows:

- o The name of each variable used in the program, along with:
  - The sequence number of the statement in which it is explicitly defined
  - The sequence numbers of all statements in which it is set
  - The sequence numbers of all statements in which it is used
- o Each label used in the program, along with:
  - The sequence number of the statement to which it is assigned
  - The sequence numbers of all statements in which it is referenced

To limit execution time and core requirements, the function shall assume the microprogram to be syntactically correct and to have no more than 5,000 labels, 2,000 variables, and 25,000 total references to labels and variables. If any source statement cannot be interpreted, the function shall report this problem in the microprogram listing, set a problem indicator, and proceed to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, set the problem indicator, then continue to process and list the source statements, storing all relevant information for which space remains.

J2.2.4 Variable Cross-Reference Function

The Variable Cross-Reference function shall produce a cross-reference for the variables in a MULTI microprogram. Inputs to the function shall consist of the information on microprogram variables recorded by the Source Code Processing function, and the column specification recorded by the User Input Processing function. The function shall alphabetize the variable names and shall produce a cross-reference incorporating all of the information on variables recorded by the Source Code Processing function. The cross-reference shall be either 80 or 132 columns in width, depending upon the column specification.

J2.2.5 Label Cross-Reference Function

The Label Cross-Reference function shall produce a cross-reference for the labels in a MULTI microprogram. Inputs to the function shall consist of the information on microprogram labels recorded by the Source Code Processing function and the column specification recorded by the User Input Processing function. The function shall alphabetize the labels and shall produce a cross-reference incorporating all of the information on labels recorded by the Source Code Processing function. The cross-reference shall be either 80 or 132 columns in width, depending upon the column specification.

J2.3 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

J2.4 Timing

There are no timing requirements on the system.

J2.5 Flexibility

The system shall be written in such a way that the limit on the number of variables, labels, and references can be easily modified.

J3. ENVIRONMENT

J3.1 Equipment Environment

See Appendix H for equipment description.

The MULTI Cross-Reference Generator will require approximately 60K words of main memory. The program will be stored on disk, requiring approximately 20K words of storage for the source code, object code, and load module. The user-prepared inputs may be entered via card reader or terminal. The MULTI source code may be stored on punched cards, magnetic tape, or disk. The output may be directed to a line printer or to a terminal. No new equipment will be required.

J3.2 Support Software Environment

See Appendix H.

J3.3 Interfaces

The program shall not interface with any other system.

J3.4 Security

The program shall have no classified components.

J3.5 Controls

No controls shall be established by the program.

J4. DESIGN DATA

J4.1 System Logical Flow

The MULTI Cross-Reference Generator shall consist of a single computer program. Figure J2 indicates the logical flow of the program and shows its inputs and outputs.

J4.2 Function Descriptions

The program shall be comprised of the five functions identified in Section J2.2. Design data for each function follow.

J4.2.1 Control Function

The Control function shall fulfill the requirements specified in Section J2.2.1. Additional design data follow.

J4.2.1.1 Inputs: The inputs to the Control function shall consist of the following:

- o The cross-reference selection values recorded by the User Input Processing function
- o The continuation option recorded by the User Input Processing function
- o The problem indicator set by the Source Code Processing function

J4.2.1.2 Outputs: The Control function shall output an explanatory message if it terminates the program prematurely.

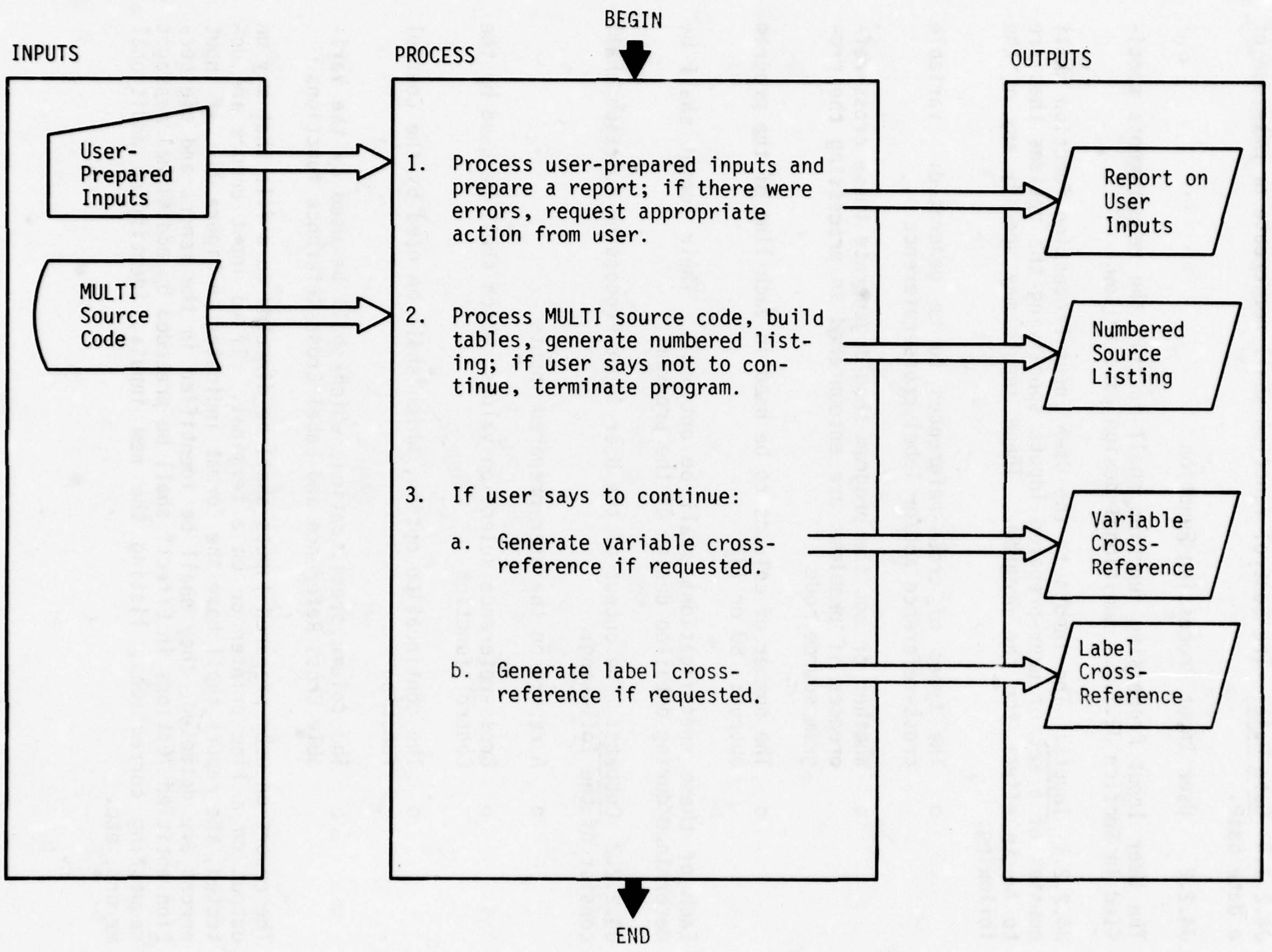

BEGIN
INPUTS
PROCESS
OUTPUTS
User-Prepared Inputs
MULTI Source Code
1. Process user-prepared inputs and prepare a report; if there were errors, request appropriate action from user.
2. Process MULTI source code, build tables, generate numbered listing; if user says not to continue, terminate program.
3. If user says to continue:
a. Generate variable cross-reference if requested.
b. Generate label cross-reference if requested.
Report on User Inputs
Numbered Source Listing
Variable Cross-Reference
Label Cross-Reference
END

Figure J2. System Logical Flow for the MULTI Cross-Reference Generator

J4.2.1.3 Data Base: The Control function shall not create or make use of a data base.

J4.2.2 User Input Processing Function

The User Input Processing function shall fulfill the requirements specified in Section J2.2.2. Additional design data follow.

J4.2.2.1 Inputs: The inputs to the User Input Processing function shall consist of a set of user-prepared inputs specifying the options that are to be in effect for the program. These inputs may specify any of the following:

- The types of cross-references to be generated: variable cross-reference and/or label cross-reference
- Whether or not the program should generate these cross-references if problems are encountered in processing the program source code
- The number of columns to be used in each line of the program output: 80 or 132

Each of these specifications shall be optional. Their format shall be determined during detailed design of the program.

J4.2.2.2 Outputs: The output of the User Input Processing function shall consist of the following:

- A report on the user prepared inputs
- Cross-reference selection values, which shall be used by the Control function
- The continuation option, which shall be used by the Control function
- The column specification, which shall be used by the Variable Cross Reference and Label Cross-Reference functions

The report on user-prepared inputs shall be directed to a file that may be output on a line printer or on a terminal. If no input errors are detected, the report shall have the format indicated in Figure J3. If input errors are detected, they shall be identified in the report and the section entitled "Options in Effect" shall be preceded by additional messages requesting corrections, listing the new inputs, identifying additional errors, etc.

```
MULTI CROSS-REFERENCE GENERATOR

USER INPUTS
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT

SELECTED CROSS-REFERENCES
         XXXXXXXXXXXXXXXXX
         XXXXXXXXXXXXXXXXX

IN CASE OF SOURCE CODE PROBLEMS:  XXXXXXXXX
NUMBER OF COLUMNS TO BE USED:  XXX
```

Figure J3. Format of the Report on User Inputs

J4.2.2.3 Data Base: The function shall not create or make use of a data base.

J4.2.3 Source Code Processing Function

The Source Code Processing function shall fulfill the requirements specified in Section J2.2.3. Additional design data follow.

J4.2.3.1 Inputs: The input to the Source Code Processing function shall consist of the source code for a microprogram written in MULTI. The MULTI source code shall be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the Nanodata Corporation report "MULTI Micromachine Description" (2 February 1976).

J4.2.3.2 Outputs: The output of the Source Code Processing function shall consist of the following:

- A numbered listing of the MULTI microprogram
- The problem indicator, which shall be used by the Control function
- The information on microprogram variables, which shall be used by the Variable Cross-Reference function
- The information on microprogram labels, which shall be used by the Label Cross-Reference function

The numbered listing shall be directed to a file that may be output on a line printer or on a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure J4. If problems are encountered, they shall be identified in the listing. The information on microprogram variables and labels shall be as specified in Section J2.2.3.

J4.2.3.3 Data Base: The MULTI microprogram to be processed may reside in a system data base. In addition, the information on microprogram variables and labels may be stored in a data base to be used by subsequent functions of the program.

J4.2.4 Variable Cross-Reference Function

The Variable Cross-Reference function shall fulfill the requirements specified in Section J2.2.4. Additional design data follow.

J4.2.4.1 Inputs: The inputs to the Variable Cross-Reference function shall consist of the following:

```
STATEMENT #             SOURCE STATEMENT

 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                             .
   .                             .
   .                             .
```

Figure J4. Format of the Numbered Source Listing

- o The information on microprogram variables recorded by the Source Code Processing function

- o The column specification recorded by the User Input Processing function

The information on microprogram variables shall be as specified in Section J2.2.3. The column specification shall indicate a value of 80 or 132.

J4.2.4.2 Outputs: The output of the function shall consist of a cross-reference for the microprogram variables. This cross-reference shall be directed to a file that may be output on a line printer or on a terminal. The format of the cross-reference shall be as indicated in Figure J5. The number of columns used in each line shall be 80 or 132, depending upon the column specification.

J4.2.4.3 Data Base: The information on microprogram variables may reside in a system data base. If so, the information need be retained only until it is used by the Variable Cross-Reference function.

J4.2.5 Label Cross-Reference Function

The Label Cross-Reference function shall fulfill the requirements specified in Section J2.2.5. Additional design data follow.

J4.2.5.1 Inputs: The inputs to the Label Cross-Reference function shall consist of the following:

- o The information on microprogram labels recorded by the Source Code Processing function

- o The column specification recorded by the User Input Processing function

The information on microprogram labels shall be as specified in Section J2.2.3. The column specification shall indicate a value of 80 or 132.

J4.2.5.2 Outputs: The output of the function shall consist of a cross-reference for the microprogram labels. This cross-reference shall be directed to a file that may be output on a line printer or on a terminal. The format of the cross-reference shall be as indicated in Figure J6. The number of columns used in each line shall be 80 or 132, depending upon the column specification.

J4.2.5.3 Data Base: The information on microprogram labels may be stored in a system data base. If so, the information need be retained only until it is used by the Label Cross-Reference function.

```
                    VARIABLE CROSS-REFERENCE

                              *** STATEMENT NUMBERS ***
VARIABLE NAME                 DEFINED    SET      REFERENCED
 XXXXXXXX                     XXXXX     XXXXX      XXXXX
                                          .          .
                                          .          .
                                          .          .
                                          .          .
 XXXXXXXX                     XXXXX     XXXXX      XXXXX
                                          .          .
                                          .          .
                                          .          .
                                          .          .
 XXXXXXXX                     XXXXX     XXXXX      XXXXX
                                          .          .
                                          .          .
                                          .          .
                                          .          .
     .                            .
     .                            .
     .                            .
```

Figure J5. Format of the Variable Cross-Reference

```
                        LABEL CROSS-REFERENCE

                             *** STATEMENT NUMBERS ***
LABEL                       DEFINED            REFERENCED
XXXXXXXX                     XXXXX         XXXXX  XXXXX  XXXXX
                                             :      :      :
XXXXXXXX                     XXXXX         XXXXX  XXXXX  XXXXX
                                             :      :      :
XXXXXXXX                     XXXXX         XXXXX  XXXXX  XXXXX
                                             :      :      :
   .                           .
   .                           .
   .                           .
```

Figure J6. Format of the Label Cross-Reference

APPENDIX K
FUNCTIONAL DESCRIPTION FOR THE SMITE GLOBAL CROSS-REFERENCE GENERATOR

K1. GENERAL

K1.1 Purpose of Functional Description

See Appendix G.

K1.2 Project References

See Appendix G.

K2. SYSTEM SUMMARY

K2.1 Background

See Appendix G.

K2.2 Objectives

The purpose of the SMITE Global Cross-Reference Generator is to provide global cross-references for the variables, labels, and processors used in a program written in SMITE. The cross-references will provide descriptive information about each entry and will identify all program statements in which the entry is defined, set, or used. These cross-references may be used for program development, testing, and maintenance.

K2.3 Existing Methods and Procedures

SMITE program development at RADC is currently performed without the aid of a global cross-reference generator. To determine the characteristics and usage of a variable, label, or processor, it is necessary for the user to inspect the source code manually.

K2.4 Proposed Methods and Procedures

The proposed method for identifying the characteristics and usage of each variable, label, and processor in a SMITE program is to submit the source code to the SMITE Global Cross-Reference Generator. Based on user-selected options, the program will provide the following:

- o A complete listing of the SMITE program, with each statement numbered
- o A global cross-reference of all variables used in the program
- o A global cross-reference of all labels used in the program

- o A global cross-reference of all processors used in the program

Figure K1 depicts the data flow for the program. Figure K2 shows the major processing steps.

K2.4.1 Summary of Improvements

During the development, testing, and maintenance of a SMITE program, it is often useful to determine exactly how and where a particular variable, label, or processor has been used. The SMITE Global Cross-Reference Generator will provide this information in a concise, accurate manner. Specific benefits to be gained from the program are as follows:

- o The information in the cross-references can be used in program debugging.
- o The cross-references can be used to determine the effects of proposed changes to the code.
- o Unreferenced labels can be identified, as can unset and unused variables.
- o The cross-references can assist in the verification of adherence to naming conventions.
- o The cross-references can serve as a permanent part of the program documentation.

K2.4.2 Summary of Impacts

See Appendix G.

K2.5 Expected Limitations

To limit execution time and core requirements, the SMITE Global Cross-Reference Generator will be designed to handle a SMITE program that is syntactically correct and that has no more than 200 processors, 5,000 labels, 2,000 variables, and 25,000 total references to processors, labels,and variables.

K3. DETAILED CHARACTERISTICS

K3.1 Specific Performance Requirements

The SMITE Global Cross-Reference Generator shall accept as input a set of user-prepared inputs and the source code of a program written in SMITE. The user-prepared inputs shall specify the following options:

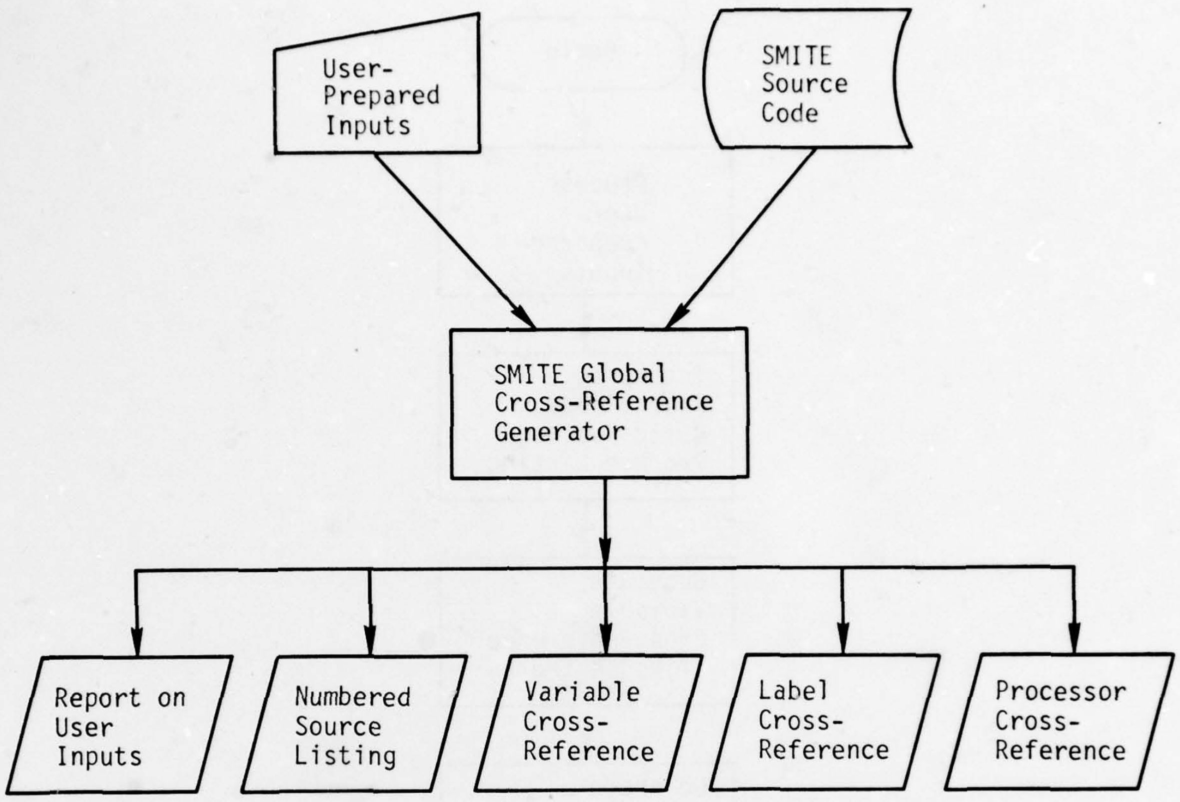


Figure K1. Data Flow for the SMITE Global Cross-Reference Generator

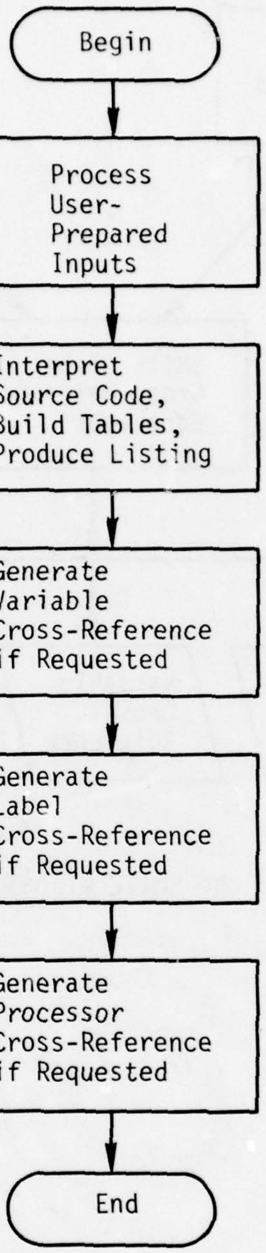

Begin
Process User-Prepared Inputs
Interpret Source Code, Build Tables, Produce Listing
Generate Variable Cross-Reference if Requested
Generate Label Cross-Reference if Requested
Generate Processor Cross-Reference if Requested
End

Figure K2. Major Processing Steps in the SMITE Global Cross-Reference Generator

- The types of cross-references to be generated (variable cross-reference, label cross-reference, and/or processor cross-reference)
- Whether or not the program should generate these cross-references if problems are encountered in processing the SMITE source code
- The number of columns to appear in each line of output (80 or 132)

Default values shall be provided for each option. A report shall be prepared identifying each input, reporting any errors detected in the inputs, and indicating the options that will be in effect for the processing of the program.

If errors are detected in the user-prepared inputs, the program shall request corrections and process the resulting inputs. When the inputs are free of errors, the program shall process each source statement in the SMITE program as follows:

- Assign the statement a sequence number
- Output a copy of the statement, along with its sequence number, in a listing of the program
- Interpret the contents of the statement and store relevant information in tables to be used for generating the global cross-references

Relevant information to be stored in the tables shall include the following:

- The name of each variable used in the microprogram, along with:
  - Its variable type
  - Its length and width
  - Its overlay characteristics
  - The processor in which it is defined
  - The sequence number of the statement in which it is explicitly defined
  - The processors and sequence numbers of all statements in which it is set

  - The processors and sequence numbers of all statements in which it is used

- Each label used in the program, along with:

  - The processor in which the label is defined

  - The sequence number of the statement to which it is assigned

  - The processors and sequence numbers of all statements in which it is referenced

- The name of each processor used in the program, along with:

  - The processor in which it is defined

  - The width of the value that it returns

  - The processors and sequence numbers of all statements in which it is called

If a source statement cannot be interpreted during the processing of the program, a message indicating this problem shall accompany the statement in the program listing and the program shall proceed to the next source statement. If any of the numerical limitations given in Section K2.5 is exceeded during the processing, the program shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

When all source statements have been processed, the program shall proceed as follows:

- If no problems were encountered during source code processing, the program shall proceed to generate the cross-references requested by the user.

- If problems were encountered, the program shall determine from user input whether to generate the requested cross-references despite the problems.

If the cross-references are not to be generated, an explanatory message shall be output and the program shall terminate. If they are to be generated, the program shall alphabetize the entries for each selected cross-reference and shall output the cross-references, incorporating the information stored in the tables during the processing of the source code and using the report width specified by the user.

K3.1.1 Accuracy and Validity

Accuracy and validity requirements are not applicable to the program.

K3.1.2 Timing

There are no timing requirements on the program.

K3.2 System Functions

The SMITE Global Cross-Reference Generator shall consist of a single program incorporating the following functions:

- o A Control function, which shall invoke the other functions in the proper sequence
- o A User Input Processing function, which shall process the user-prepared inputs
- o A Source Code Processing function, which shall interpret each source statement, assign it a sequence number, produce a source listing, and build the tables needed for cross-reference generation
- o A Variable Cross-Reference function, which shall generate the variable cross-reference
- o A Label Cross-Reference function, which shall generate the label cross-reference
- o A Processor Cross-Reference function, which shall generate the processor cross-reference

These six functions suffice to satisfy the performance requirements identified in Section K3.1.

K3.3 Inputs/Outputs

K3.3.1 Inputs

The inputs to the SMITE Global Cross-Reference Generator shall consist of user-prepared inputs and the source code of a program written in SMITE. The user-prepared inputs shall be as follows:

- o Specification of the types of cross-references desired: variable cross-reference, label cross-reference, and/or processor cross-reference
- o Specification of whether or not to generate the selected cross-references if problems are encountered during processing of the SMITE source code

- o Specification of the number of columns that are to be used in each line of the program outputs: 80 or 132

These specifications may be input via card reader or terminal. Default values shall be provided for any missing specifications. The SMITE source code may be entered via punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image.

K3.3.2 Outputs

The output of the program shall consist of the following:

- o A report on the user-prepared inputs
- o A numbered listing of the SMITE source code
- o A variable cross-reference, if requested
- o A label cross-reference, if requested
- o A processor cross-reference, if requested

The formats of these outputs shall be as indicated in Figures K3, K4, K5, K6, and K7, respectively. The reports shall be directed to a line printer or to a terminal.

K3.4 Data Characteristics

The SMITE program being processed may be contained in a system data base.

K3.5 Failure Contingencies

Not applicable.

K4. ENVIRONMENT

K4.1 Equipment Environment

The SMITE Global Cross-Reference Generator will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 80K words of main memory to operate. The program will be stored on disk. The user-prepared inputs may be entered via card reader or terminal. The SMITE source code will be a sequential file that may be stored on punched cards, magnetic tape, or disk. The output will be directed to a line printer or to a terminal. No new equipment will be required.

K4.2 Support Software Environment

See Appendix G.

K4.3 Interfaces

The program will not interface with any other system.

```
SMITE GLOBAL CROSS-REFERENCE GENERATOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


OPTIONS IN EFFECT:


SELECTED CROSS-REFERENCES:
     XXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXX

IN CASE OF SOURCE CODE PROBLEMS:  XXXXXXXX

NUMBER OF COLUMNS TO BE USED:  XXX
```

Figure K3. Format of the Report on User Inputs

```
STATEMENT #          SOURCE STATEMENT

  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    .                     .
    .                     .
    .                     .
```

Figure K4. Format of the Numbered Source Listing

```
VARIABLE CROSS-REFERENCE

                                                            ******** DEFINITION *******           ************** USAGE ***************
VARIABLE-NAME        LENGTH       WIDTH        CLASS        PROCESSOR                  STMT           PROCESSOR                          STATEMENTS
XXXXXXXXXXXXXXXXXXXX <XXX:XXX>    <XX:XX>      XXXXX        XXXXXXXXXXXXXXXXXXXX       XXXXX    SET:  XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                        XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                                  .
                                                                                                                  .
                                                                                                                  .
                                                                                                  REF:  XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                        XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                                  .
                                                                                                                  .
                                                                                                                  .
                                                                                                  OVERLAY:  XXXXXXXXXXXXXXXXXXXX
                                                                                                            XXXXXXXXXXXXXXXXXXXX
                                                                                                                      .
                                                                                                                      .
                                                                                                                      .

XXXXXXXXXXXXXXXXXXXX <XXX:XXX>    <XX:XX>      XXXXX        XXXXXXXXXXXXXXXXXXXX       XXXXX    SET:  XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                        XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                                  .             XXXXX    XXXXX
                                                                                                                  .             XXXXX    XXXXX
                                                                                                                  .
                                                                                                  REF:  XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                        XXXXXXXXXXXXXXXXXXXX    XXXXX    XXXXX
                                                                                                                  .
                                                                                                                  .
                                                                                                                  .
                                                                                                  OVERLAY:  XXXXXXXXXXXXXXXXXXXX
                                                                                                            XXXXXXXXXXXXXXXXXXXX
                                                                                                                      .
                                                                                                                      .
                                                                                                                      .
```

Figure K5. Format of the Variable Cross-Reference

```
LABEL CROSS-REFERENCE

                     *********** DEFINITION ***********     ****************************** USAGE ******************************
LABEL                PROCESSOR                STATEMENTS    PROCESSOR                STATEMENTS
XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXX XXXXX XXXXX      XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                           XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                                     .
                                                                     .
                                                                     .
XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXX XXXXX XXXXX      XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                           XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                                     .
                                                                     .
                                                                     .
XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXX XXXXX XXXXX      XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                           XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                                     .
                                                                     .
                                                                     .
XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXX XXXXX XXXXX      XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                           XXXXXXXXXXXXXXXXXXXX   XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
                                                                     .
                                                                     .
                                                                     .
```

Figure K6. Format of the Label Cross-Reference

```
PROCESSOR CROSS-REFERENCE

                      ******* DEFINITION ********    RETURN      *************************** USAGE **************************
PROCESSOR             PROCESSOR             STMT     WIDTH       PROCESSOR                STATEMENTS
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX    <XX:XX>     XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                 XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                           .
                                                                           .
                                                                           .

XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX    <XX:XX>     XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                 XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                           .
                                                                           .
                                                                           .

XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX    <XX:XX>     XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                 XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                           .
                                                                           .
                                                                           .

XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX    <XX:XX>     XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                 XXXXXXXXXXXXXXXXXXXX     XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                           .
                                                                           .
                                                                           .
```

Figure K7. Format of the Processor Cross-Reference

This System Specification is written to fulfill the following objectives:

- o To provide detailed definition of the system functions
- o To communicate details of the on-going analysis to the user's operational personnel
- o To define in detail the interfaces with other systems and subsystems and the facilities to be utilized for accomplishing the interfaces

K4.4 Security

The program will have no classified components.

K5. COST FACTORS

Development of the SMITE Global Cross-Reference Generator will require approximately 8 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the program.

K6. DEVELOPMENTAL PLAN

The SMITE Global Cross-Reference Generator is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. *The overall development schedule will depend upon the tools* selected. The recommended schedule for development of the SMITE Global Cross-Reference Generator is given in Figure K8.

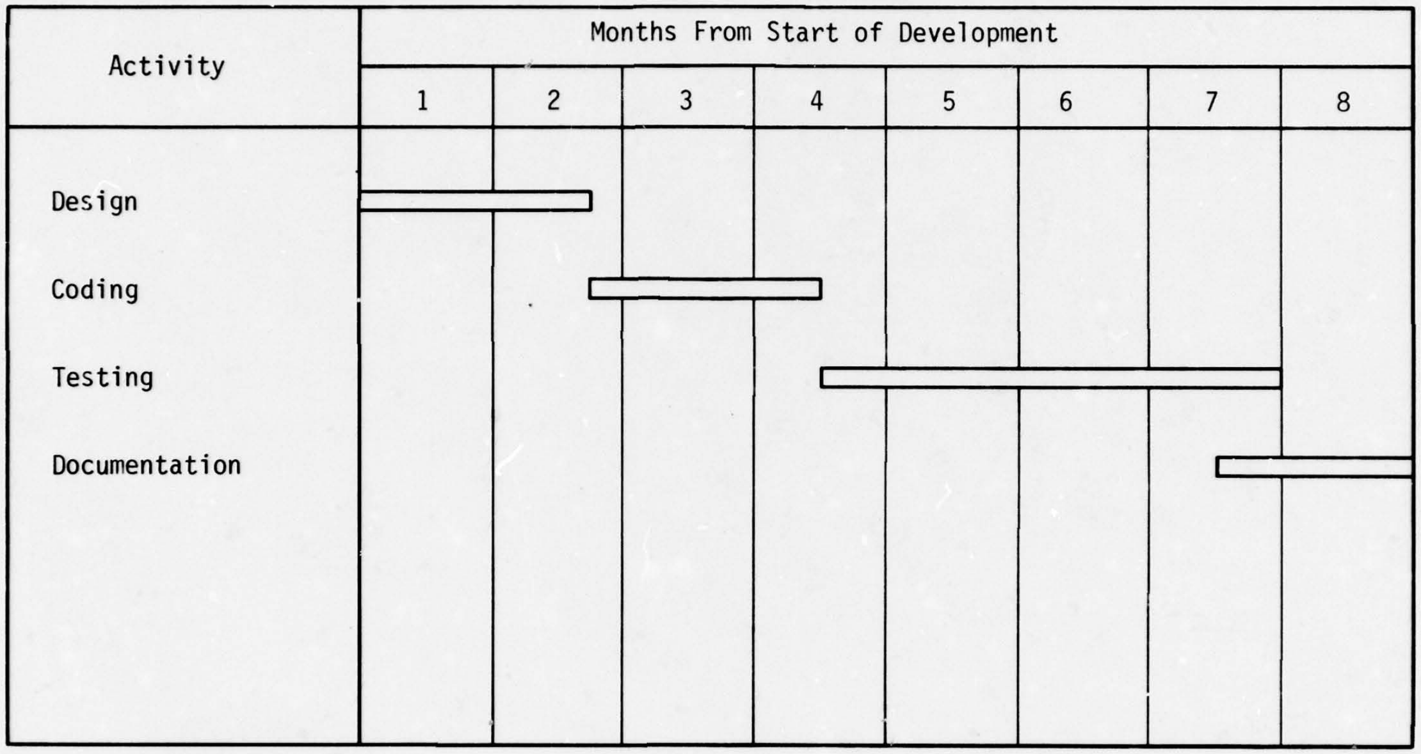


Figure K8. Development Schedule for the SMITE Global Cross-Reference Generator

APPENDIX L
SYSTEM SPECIFICATION FOR THE SMITE GLOBAL CROSS-REFERENCE GENERATOR

L1. GENERAL

L1.1 Purpose of the System Specification

See Appendix H.

L1.2 Project References

See Appendix H.

L2. SUMMARY OF REQUIREMENTS

L2.1 System Description

The SMITE Global Cross-Reference Generator shall provide global cross-references for the variables, labels, and processors in a program written in SMITE. The cross-reference shall provide descriptive information about each entry and shall identify all processors and program statements in which the entry is defined, set, or used. The system shall consist of a single program. Figure L1 illustrates the relationship of the program's components.

L2.2 System Functions

The SMITE Global Cross-Reference Generator shall be comprised of the following functions:

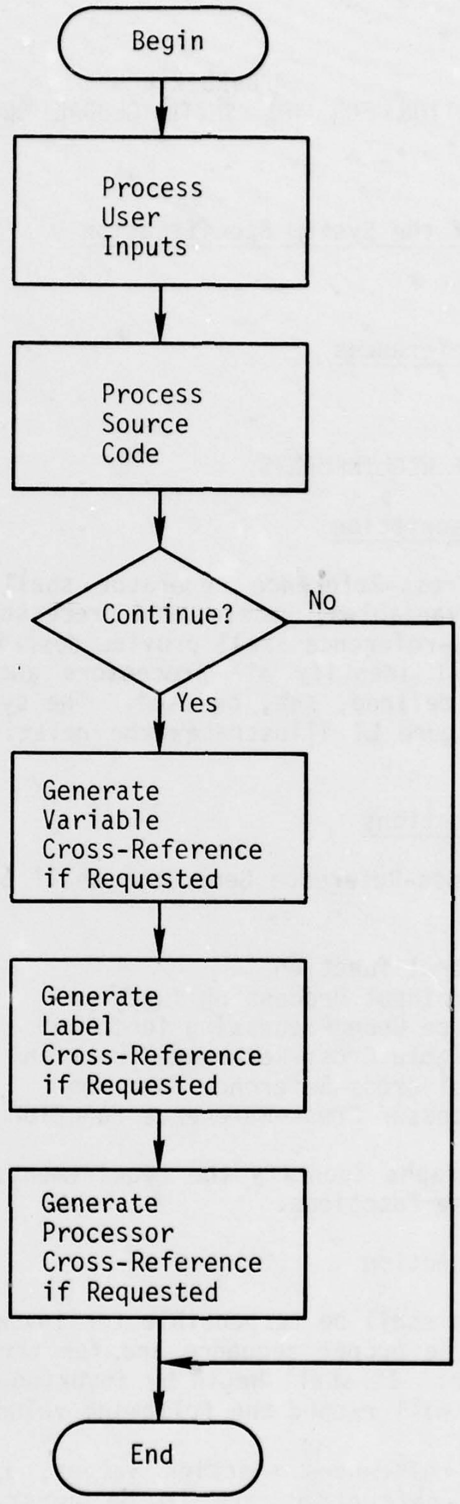
- o A Control function
- o A User Input Processing function
- o A Source Code Processing function
- o A Variable Cross-Reference function
- o A Label Cross-Reference function
- o A Processor Cross-Reference function

The following paragraphs identify the requirements that are to be satisfied by each of these functions.

L2.2.1 Control Function

The Control function shall be responsible for invoking the other functions of the program in the proper sequence and for terminating the program at the appropriate time. It shall begin by invoking the User Input Processing function, which will record the following values:

- o Cross-reference selection values, indicating which of the cross-references are to be generated (variable cross-reference, label cross-reference, and/or processor cross-reference)



Begin
Process User Inputs
Process Source Code
Continue?
No
Yes
Generate Variable Cross-Reference if Requested
Generate Label Cross-Reference if Requested
Generate Processor Cross-Reference if Requested
End

Figure L1. Relationship of the Components of the SMITE Global Cross-Reference Generator

- A "continuation option," indicating whether these cross-references are to be generated if problems are encountered in processing the source code

The Control function shall next invoke the Source Code Processing function. This function will set a value indicating whether problems were encountered in processing the source code. If problems were encountered and the continuation option indicates that the program should not continue, the Control function shall output an explanatory message and terminate the program. If no problems were encountered, or if problems were encountered but the continuation option indicates that the program should go on, the Control function shall use the cross-reference selection values to determine which of the following functions to invoke:

- The Variable Cross-Reference function
- The Label Cross-Reference function
- The Processor Cross-Reference function

The appropriate functions shall be invoked, and when they have completed their processing the Control function shall terminate the program.

L2.2.2 User Input Processing Function

The User Input Processing function shall accept as input a set of user-prepared inputs specifying program options. The options that may be specified shall be as follows:

- The cross-references to be generated (variable cross-reference, label cross-reference, and/or processor cross-reference)

- Whether the program should generate these cross-references if problems are encountered in processing the SMITE source code

- The number of columns to be used in each line of program output (80 or 132)

Each of these specifications shall be optional; default options shall be provided as follows:

- Cross-references to be generated: All three
- Program action if problems are encountered: Continue
- Number of columns: 132

The function shall check the inputs for errors in format and content and shall record the options specified, providing default values as required. The function shall prepare a report listing each user input and identifying any errors detected. If errors are present, the function shall request corrections, then process and list the corrected inputs as it did the original ones. When all of the inputs have been successfully processed, the function shall conclude the report by identifying the options that are to be in effect for the program.

L2.2.3 Source Code Processing Function

The Source Code Processing function shall accept as input the source code of a program written in SMITE. The function shall process each source statement of the program as follows:

- Assign the statement a sequence number
- Output a copy of the statement, along with its sequence number, in a listing of the program
- Interpret the contents of the statement and store relevant information in tables to be used for generating the global cross-references

Source code comments shall be copied into the program listing without being interpreted or assigned a sequence number.

The information to be stored in tables shall be as follows:

- The name of each variable used in the program, along with:
  - Its variable type
  - Its length and width
  - Its overlay characteristics
  - The processor in which it is defined
  - The sequence number of the statement in which it is explicitly defined
  - The processors and sequence numbers of all statements in which it is set
  - The processors and sequence numbers of all statements in which it is used
- Each label used in the program, along with:
  - The processor in which the label is defined
  - The sequence number of the statement to which it is assigned

- The processors and sequence numbers of all statements in which it is referenced

o The name of each processor used in the program, along with:

  - The processor in which it is defined
  - The width of the value that it returns
  - The processors and sequence numbers of all statements in which it is called

To limit execution time and core requirements, the function shall assume the program to be syntactically correct and to have no more than 200 processors, 5,000 labels, 2,000 variables, and 25,000 total references to processors, labels, and variables. If any source statement cannot be interpreted, the function shall report this problem in the program listing, set a problem indicator, and proceed to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, set the problem indicator, then continue to process and list the source statements, storing all relevant information for which space remains.

L2.2.4 Variable Cross-Reference Function

The Variable Cross-Reference function shall generate a global cross-reference for the variables in a SMITE program. Inputs to the function shall consist of the information on program variables recorded by the Source Code Processing function and the column specification recorded by the User Input Processing function. The function shall alphabetize the variable names and shall produce a cross-reference incorporating all of the information on variables recorded by the Source Code Processing function. The cross-reference shall be either 80 or 132 columns in width, depending upon the column specification.

L2.2.5 Label Cross-Reference Function

The Label Cross-Reference function shall be responsible for generating a global cross-reference for the labels in a SMITE program. Inputs to the function shall consist of the information on program labels recorded by the Source Code Processing function and the column specification recorded by the User Input Processing function. The function shall alphabetize the labels and shall produce a cross-reference incorporating all of the information on labels recorded by the Source Code Processing function. The cross-reference shall be either 80 or 132 columns in width, depending upon the column specification.

L2.2.6 Processor Cross-Reference Function

The Processor Cross-Reference function shall be responsible for generating a global cross-reference for the processors in a SMITE program. Inputs to the function shall consist of the information on processors recorded by the Source Code Processing function and the column specification recorded by the User Input Processing function. The function shall alphabetize the processor names and shall produce a cross-reference incorporating all of the information on processors recorded by the Source Code Processing function. The cross-reference shall be either 80 or 132 columns in width, depending upon the column specification.

L2.3 Accuracy and Validity

Accuracy and validity requirements are not applicable to the program.

L2.4 Timing

There are no timing requirements on the program.

L2.5 Flexibility

The program shall be written in such a way that the limit on the number of processors, labels, variables, and references can be easily modified.

L3. ENVIRONMENT

L3.1 Equipment Environment

See Appendix H for equipment description.

The SMITE Global Cross-Reference Generator will require approximately 80K words of main memory. The program will be stored on disk, requiring approximately 30K words of storage for the source code, object code, and load module. The user-prepared inputs may be entered via card reader or terminal. The SMITE source code may be stored on punched cards, magnetic tape, or disk. The output may be directed to a line printer or to a terminal. No new equipment will be required.

L3.2 Support Software Environment

See Appendix H.

L3.3 Interfaces

The program shall not interface with any other system.

L3.4 Security

The program shall have no classified components.

L3.5 Controls

No controls shall be established by the program.

L4. DESIGN DATA

L4.1 System Logical Flow

The SMITE Global Cross-Reference Generator shall consist of a single computer program. Figure L2 indicates the logical flow of the program and shows the program's inputs and outputs.

L4.2 Function Descriptions

The program shall be comprised of the six functions identified in Section L2.2. Design data for each function follow.

L4.2.1 Control Function

The Control function shall fulfill the requirements specified in Section L2.2.1. Additional design data follow.

L4.2.1.1 Inputs: The inputs to the Control function shall consist of the following:

- o The cross-reference selection values recorded by the User Input Processing function
- o The continuation option recorded by the User Input Processing function
- o The problem indicator set by the Source Code Processing function

L4.2.1.2 Outputs: Output from the Control function shall consist of an explanatory message to appear at the conclusion of the numbered listing if the program is terminated prematurely.

L4.2.1.3 Data Base: The Control function shall not create or make use of a data base.

L4.2.2. User Input Processing Function

The User Input Processing function shall fulfill the requirements specified in Section L2.2.2. Additional design data follow.

L4.2.2.1 Inputs: The inputs to the User Input Processing function shall consist of a set of user-prepared inputs specifying the options that are to be in effect for the program. These inputs may specify any of the following:

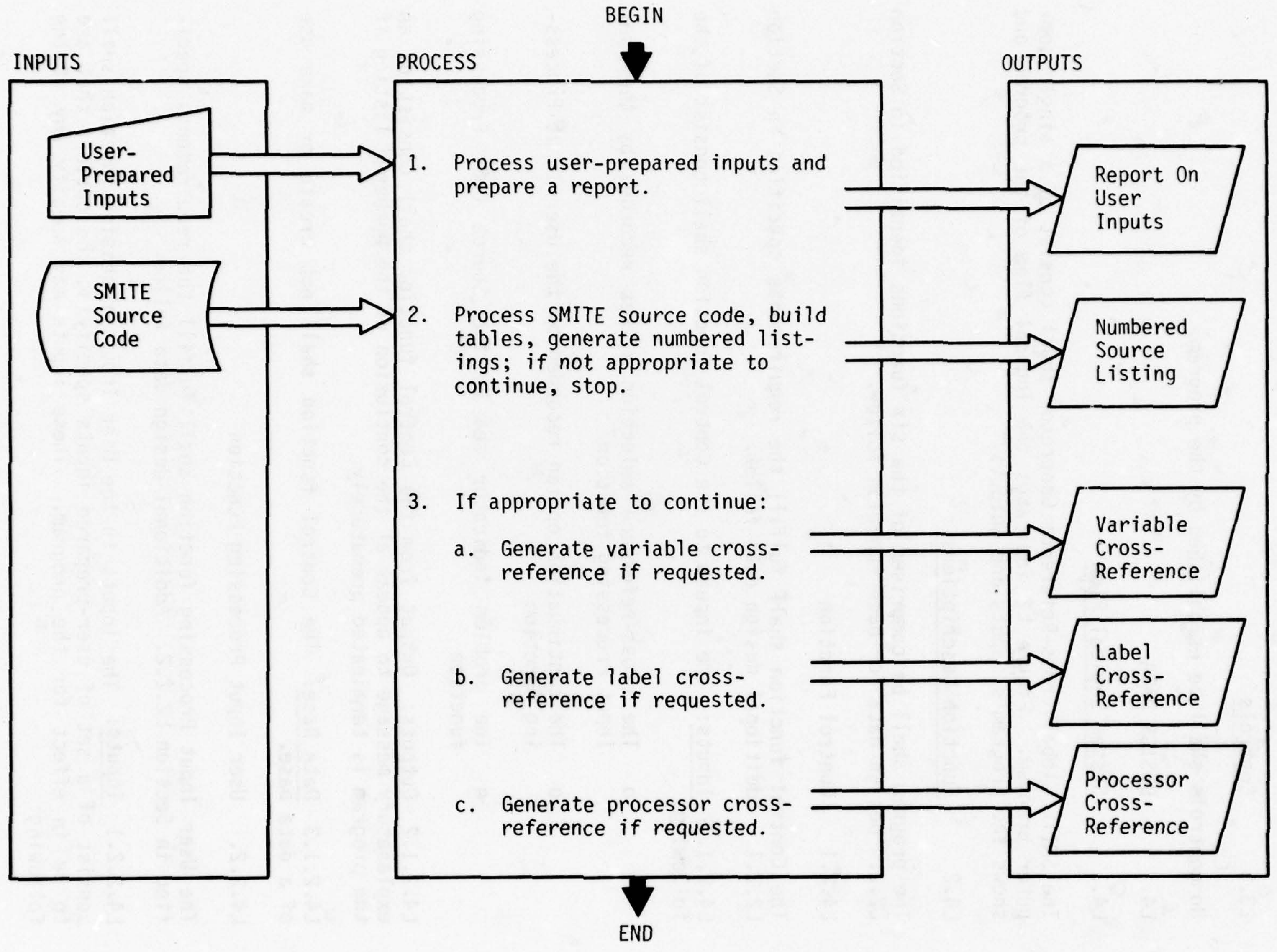

BEGIN
INPUTS
PROCESS
OUTPUTS
User-Prepared Inputs
SMITE Source Code
1. Process user-prepared inputs and prepare a report.
2. Process SMITE source code, build tables, generate numbered listings; if not appropriate to continue, stop.
3. If appropriate to continue:
a. Generate variable cross-reference if requested.
b. Generate label cross-reference if requested.
c. Generate processor cross-reference if requested.
Report On User Inputs
Numbered Source Listing
Variable Cross-Reference
Label Cross-Reference
Processor Cross-Reference
END

Figure L2. System Logical Flow for the SMITE Global Cross-Reference Generator

- The types of cross-references to be generated: variable cross-reference, label cross-reference, and/or processor cross-reference

- Whether or not the program is to generate these cross-references if problems are encountered in processing the program source code

- The number of columns to be used in each line of the program output: 80 or 132

Each of these specifications shall be optional. Their formats shall be determined during detailed design of the program.

L4.2.2.2 Outputs: The output of the User Input Processing function shall consist of the following:

- A report on the user-prepared inputs

- Cross-reference selection values, which shall be used by the Control function

- The continuation option, which shall be used by the Control function

- The column specification, which shall be used by the Variable Cross Reference, Label Cross-Reference, and Processor Cross-Reference functions

The report on user-prepared inputs shall be directed to a file that may be output on a line printer or a terminal. If no input errors are detected, the report shall have the format indicated in Figure L3. If input errors are detected, they shall be identified in the report and the section entitled "Options in Effect" shall be preceded by additional messages requesting corrections, listing the new inputs, identifying additional errors, etc.

L4.2.2.3 Data Base: The function shall not create or make use of a data base.

L4.2.3 *Source Code Processing Function*

The Source Code Processing function shall fulfill the requirements specified in Section L2.2.3. Additional design data follow.

L4.2.3.1 Inputs: The input to the Source Code Processing function shall consist of the source code for a program written in SMITE. The SMITE source code shall be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

```
SMITE GLOBAL CROSS-REFERENCE GENERATOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


OPTIONS IN EFFECT:


SELECTED CROSS-REFERENCES:
     XXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXX

IN CASE OF SOURCE CODE PROBLEMS:  XXXXXXXX

NUMBER OF COLUMNS TO BE USED:  XXX
```

Figure L3. Format of the Report on User Inputs

L4.2.3.2 Outputs: The output of the Source Code Processing function shall consist of the following:

- o A numbered listing of the SMITE program
- o The problem indicator, which shall be used by the Control function
- o The information on program variables, which shall be used by the Variable Cross-Reference function
- o The information on program labels, which shall be used by the Label Cross-Reference function
- o The information on program processors, which shall be used by the Processor Cross-Reference function

The numbered listing shall be directed to a file that may be output on a line printer or a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure L4. If problems are encountered, they shall be identified in the listing.

The information on program variables, labels, and processors shall be as specified in Section L2.2.3.

L4.2.3.3 Data Base: The SMITE program to be processed may be contained in a system data base. In addition, the information on program variables, labels, and processors may be stored in a data base for use by subsequent functions of the program.

L4.2.4 Variable Cross-Reference Function

The Variable Cross-Reference function shall fulfill the requirements specified in Section L2.2.4. Additional design data follow.

L4.2.4.1 Inputs: The inputs to the Variable Cross-Reference function shall consist of the following:

- o The information on program variables recorded by the Source Code Processing function
- o The column specification recorded by the User Input Processing function

The information on program variables shall be as specified in Section L2.2.3. The column specification shall indicate the value 80 or 132.

L4.2.4.2 Outputs: The output of the function shall consist of a global cross-reference for the program variables. This cross-reference shall be directed to a file that may be output on a line printer or on a terminal.

```
STATEMENT #           SOURCE STATEMENT

 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                       .
   .                       .
   .                       .
```

Figure L4. Format of the Numbered Source Listing

The format of the cross-reference shall be as indicated in Figure L5. The number of columns used in each line of the cross-reference shall be 80 or 132, depending upon the column specification.

L4.2.4.3 Data Base: The information on program variables may reside in a system data base. This information needs to be retained only until it is used by the Variable Cross-Reference function.

L4.2.5 Label Cross-Reference Function

The Label Cross-Reference function shall fulfill the requirements specified in Section L2.2.5. Additional design data follow.

L4.2.5.1 Inputs: The inputs to the Label Cross-Reference function shall consist of the following:

- o The information on program labels recorded by the Source Code Processing function
- o The column specification recorded by the User Input Processing function

The information on program labels shall be as specified in Section L2.2.3. The column specification shall indicate the value 80 or 132.

L4.2.5.2 Outputs: The output of the function shall consist of a global cross-reference for the program labels. This cross-reference shall be directed to a file that may be output on a line printer or on a terminal. The format of the cross-reference shall be as indicated in Figure L6. The number of columns used in each line of the cross-reference shall be 80 or 132, depending upon the column specification.

L4.2.5.3 Data Base: The information on program labels may be stored in a system data base. This information need be retained only until it is used by the Label Cross-Reference function.

L4.2.6 Processor Cross-Reference Function

The Processor Cross-Reference function shall fulfill the requirements specified in Section L2.2.6. Additional design data follow.

L4.2.6.1 Inputs: The inputs to the Processor Cross-Reference function shall consist of the following:

- o The information on processors recorded by the Source Code Processing function
- o The column specification recorded by the User Input Processing function

```
VARIABLE CROSS-REFERENCE

                                                     ******** DEFINITION *******                ************** USAGE **************
VARIABLE-NAME        LENGTH       WIDTH      CLASS   PROCESSOR                STMT              PROCESSOR                  STATEMENTS

XXXXXXXXXXXXXXXXXXXX <XXX:XXX>    <XX:XX>    XXXXX   XXXXXXXXXXXXXXXXXXXX     XXXXX    SET:     XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                          .
                                                                                                          .
                                                                                                          .
                                                                                       REF:     XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                          .
                                                                                                          .
                                                                                                          .
                                                                                       OVERLAY:     XXXXXXXXXXXXXXXXXXXX
                                                                                                    XXXXXXXXXXXXXXXXXXXX
                                                                                                              .
                                                                                                              .
                                                                                                              .

XXXXXXXXXXXXXXXXXXXX <XXX:XXX>    <XX:XX>    XXXXX   XXXXXXXXXXXXXXXXXXXX     XXXXX    SET:     XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                                          XXXXX     XXXXX
                                                                                                                          XXXXX     XXXXX
                                                                                                          .
                                                                                                          .
                                                                                       REF:     XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                XXXXXXXXXXXXXXXXXXXX      XXXXX     XXXXX
                                                                                                          .
                                                                                                          .
                                                                                                          .
                                                                                       OVERLAY:     XXXXXXXXXXXXXXXXXXXX
                                                                                                    XXXXXXXXXXXXXXXXXXXX
                                                                                                              .
                                                                                                              .
                                                                                                              .
```

Figure L5. Format of the Variable Cross-Reference

```
LABEL CROSS-REFERENCE

                      *********** DEFINITION ***********     ****************************** USAGE *******************************
LABEL                 PROCESSOR                 STATEMENTS     PROCESSOR                  STATEMENTS
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX  XXXXX     XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                             XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                      .
                                                                      .
                                                                      .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX  XXXXX     XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                             XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                      .
                                                                      .
                                                                      .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX  XXXXX     XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                             XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                      .
                                                                      .
                                                                      .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX  XXXXX     XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                             XXXXXXXXXXXXXXXXXXXX  XXXXX   XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                      .
                                                                      .
                                                                      .
```

Figure L6. Format of the Label Cross-Reference

The information on processors shall be as specified in Section L2.2.3. The column specification shall indicate the value 80 or 132.

L4.2.6.2 Outputs: The output of the function shall consist of a global cross-reference for the processors. This cross-reference shall be directed to a file that may be output on a line printer or on a terminal. The format of the cross-reference shall be as indicated in Figure L7. The number of columns used in each line of the cross-reference shall be 80 or 132, depending upon the column specification.

L4.2.6.3 Data Base: The information on processors may be stored in a system data base. This information need be retained only until it is used by the Processor Cross-Reference function.

```
PROCESSOR CROSS-REFERENCE

                      ******* DEFINITION ********   RETURN     *************************** USAGE *************************
PROCESSOR             PROCESSOR             STMT    WIDTH      PROCESSOR               STATEMENTS
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX   <XX:XX>    XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                               XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                         .
                                                                         .
                                                                         .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX   <XX:XX>    XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                               XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                         .
                                                                         .
                                                                         .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX   <XX:XX>    XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                               XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                         .
                                                                         .
                                                                         .
XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXX   <XX:XX>    XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                               XXXXXXXXXXXXXXXXXXXX    XXXXX   XXXXX   XXXXX   XXXXX   XXXXX
                                                                         .
                                                                         .
                                                                         .
```

Figure L7. Format of the Processor Cross-Reference

APPENDIX M
FUNCTIONAL DESCRIPTION FOR THE CONTROL FLOW ANALYZER

M1. GENERAL

M1.1 Purpose of Functional Description

See Appendix G.

M1.2 Project References

See Appendix G.

M2. SYSTEM SUMMARY

M2.1 Background

See Appendix G.

M2.2 Objectives

The purpose of the Control Flow Analyzer is to provide an automated means of investigating the flow of control within a microprogram written in SMITE or MULTI. The system will be capable of identifying unreachable code, generating a flowchart of the microprogram, and responding interactively to user questions concerning the flow of control within the microprogram. It is intended to be used to identify logical errors in microprograms, to provide a part of the microprogram documentation, and to aid in the testing and maintenance of the microprogram.

M2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of a Control Flow Analyzer. Identification of unreachable code, generation of microprogram flowcharts, and analysis of microprogram flow of control are manual processes.

M2.4 Proposed Methods and Procedures

The proposed method for investigating the flow of control within a microprogram is to submit the microprogram to the Control Flow Analyzer. The system will operate in three phases. Phase 1 will analyze the SMITE or MULTI source code, generate a nominal internal model of the microprogram (i.e., a model of the microprogram as it was written), and produce a numbered microprogram listing, a report on user inputs, and a report identifying any unreachable code. Phase 2 will build a structured version of the internal model. Phase 3 will use the internal models to generate a nominal or structured version of the microprogram and to respond interactively to user questions about the microprogram's flow of control. Figure M1 depicts the data flow for the system. Figure M2 shows its major processing steps.

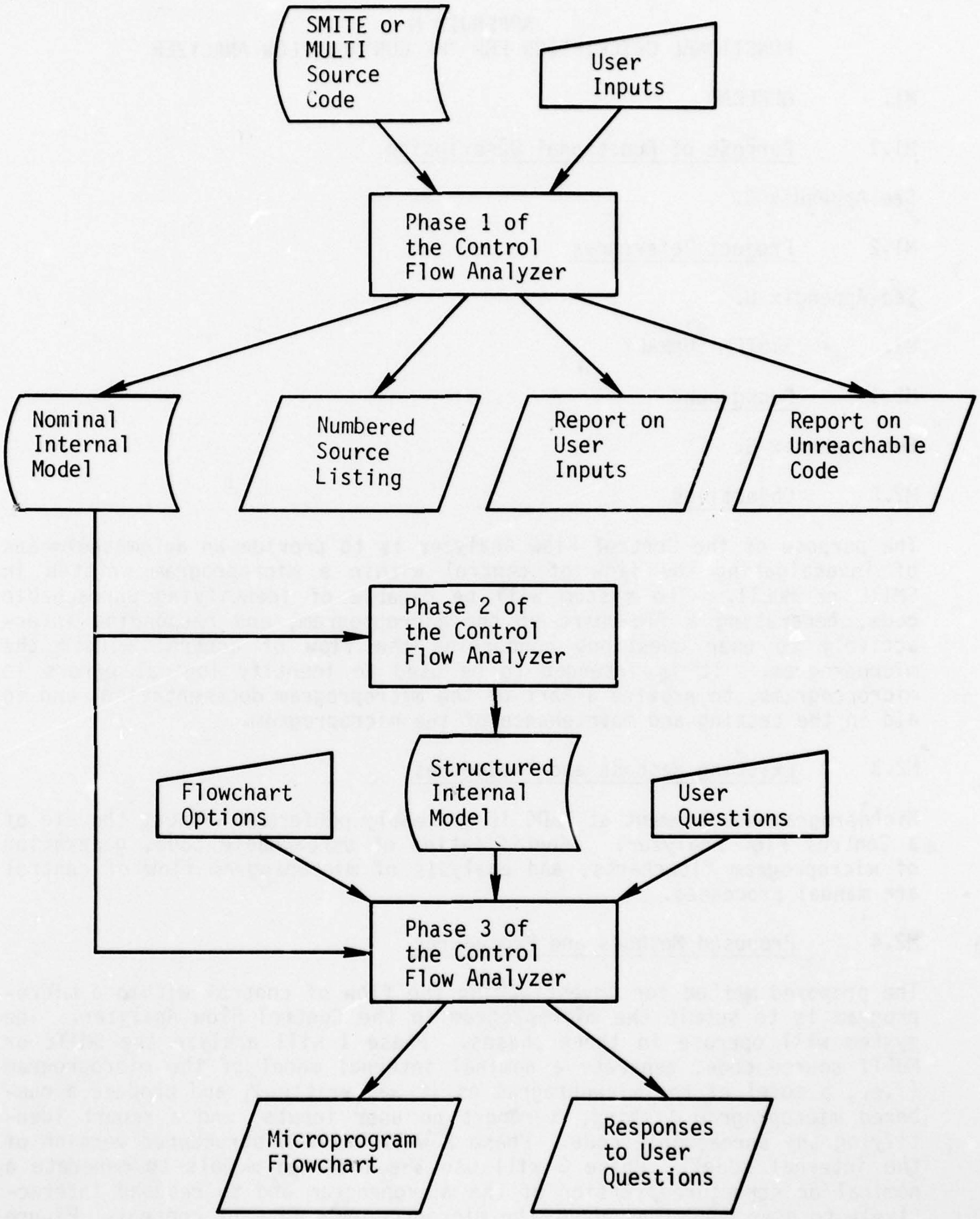

SMITE or MULTI Source Code
User Inputs
Phase 1 of the Control Flow Analyzer
Nominal Internal Model
Numbered Source Listing
Report on User Inputs
Report on Unreachable Code
Phase 2 of the Control Flow Analyzer
Flowchart Options
Structured Internal Model
User Questions
Phase 3 of the Control Flow Analyzer
Microprogram Flowchart
Responses to User Questions

Figure M1. Data Flow for the Control Flow Analyzer

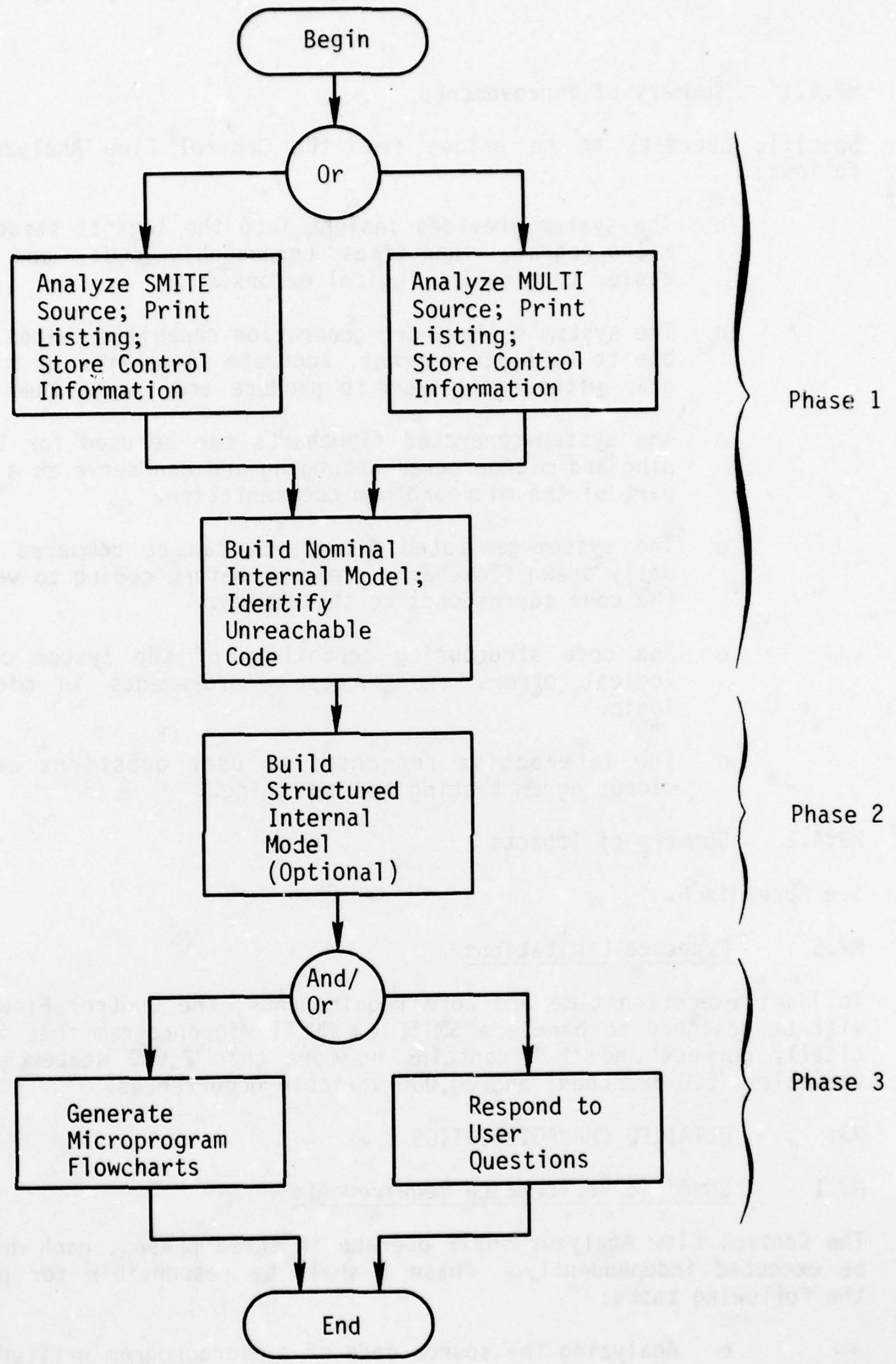

Begin
Or
Analyze SMITE Source; Print Listing; Store Control Information
Analyze MULTI Source; Print Listing; Store Control Information
Phase 1
Build Nominal Internal Model; Identify Unreachable Code
Build Structured Internal Model (Optional)
Phase 2
And/ Or
Generate Microprogram Flowcharts
Respond to User Questions
Phase 3
End

Figure M2. Major Processing Steps in the Control Flow Analyzer

M2.4.1 Summary of Improvements

Specific benefits to be gained from the Control Flow Analyzer are as follows:

- o The system provides insight into the logical structure of a microprogram, identifies unreachable code, and makes it easier to pinpoint logical errors.
- o The system's flowchart generation capability makes it possible to maintain current, accurate flowcharts of a microprogram without the need to produce and update them manually.
- o The system-generated flowcharts can be used for test planning and microprogram debugging and can serve as a permanent part of the microprogram documentation.
- o The system-generated flowcharts can be compared with manually drawn flowcharts prepared before coding to verify that the code corresponds to the design.
- o The code structuring capability of the system can reveal logical errors and suggest improvements in microprogram logic.
- o The interactive responses to user questions can aid in microprogram testing and debugging.

M2.4.2 Summary of Impacts

See Appendix G.

M2.5 <u>Expected Limitations</u>

To limit execution time and core requirements, the Control Flow Analyzer will be designed to handle a SMITE or MULTI microprogram that is syntactically correct and that contains no more than 2,000 statements, 2,000 variables, 500 branches, and 20,000 variable occurrences.

M3. DETAILED CHARACTERISTICS

M3.1 <u>Specific Performance Requirements</u>

The Control Flow Analyzer shall operate in three phases, each designed to be executed independently. Phase 1 shall be responsible for performing the following tasks:

- o Analyzing the source code of a microprogram written in SMITE or MULTI

- o Assigning each source statement a sequence number and generating a numbered listing of the microprogram
- o Building a nominal internal model of the microprogram (i.e., a model of the microprogram as it was written)
- o Producing a report identifying any unreachable code in the microprogram

Phase 1 shall operate in two steps. In the first step, the source code shall be analyzed, the numbered listing shall be generated, and control flow information shall be stored in tables. If any source statement cannot be interpreted during this step, it shall be identified in the numbered listing, and the system shall proceed to the next source statement. If any of the numerical limitations given in Section M2.5 is exceeded during the processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

In the second step of Phase 1, the system shall build a nominal internal model of the microprogram and shall generate a report that identifies any unreachable code. For a SMITE program, this step shall proceed directly from the first without the need for additional inputs. For a MULTI microprogram, this step will require user-supplied inputs specifying all possible destinations for indirect branches. These user inputs shall make use of the sequence numbers provided in the numbered listing. The system shall process the user inputs, check them for errors in format and content, and prepare a report on them, identifying any errors detected. If errors are present, the system shall permit the user to make corrections interactively. When the inputs are correct, the system shall use them to aid in building the nominal internal model. This model shall indicate all possible successors to each source statement and shall contain the text of each statement. It shall be used to generate the report that identifies any source statements that cannnot be reached during execution of the microprogram and shall be stored on disk to be used as input to Phases 2 and 3 of the system.

In Phase 2, which shall be optional, the system shall generate a structured version of the internal model. All flow of control in this model shall adhere to structured programming concepts. This model shall be stored on disk to be used during Phase 3 for generating a structured flowchart of the microprogram and for generating interactive responses to user questions about the microprogram flow of control.

In Phase 3, the system shall use the nominal and/or structured internal models of the microprogram to generate control flow information for the user. Three types of output shall be provided:

- o A nominal flowchart of the microprogram, i.e., a flowchart of the microprogram as it was written

- o A structured flowchart of the microprogram
- o Interactive responses to user questions about the microprogram's flow of control

Each of these outputs shall be optional. The flowcharts shall be generated according to user-specified options that include the following:

- o Output device (printer, plotter, or CRT)
- o Type of flowchart symbols to be used
- o Size of pages, borders, and characters
- o Collapsing of consecutive process boxes
- o Appending of labels and prologues to the flowchart
- o Output of a directory of connector occurrences
- o Substitution of mathematical notation in source text
- o Substitution of sequence numbers for source text
- o Segmentation of the flowchart
- o Structuring of the flowchart

The resulting flowchart shall be a detailed, source-level flowchart of the microprogram.

Interactive responses shall be generated for the following types of user questions:

- o What statements might immediately follow statement N during microprogram execution?
- o What statements might immediately precede statement N during microprogram execution?
- o What are all possible paths from statement M to statement N?

Statements shall be identified by the sequence numbers assigned to them in the numbered listing. In enumerating microprogram paths, the system shall limit itself to a single execution of each loop.

M3.1.1 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

M3.1.2 Timing

There are no timing requirements on the system.

M3.2 System Functions

The Control Flow Analyzer shall perform six major functions, each of which shall be implemented as a separate program in the system. The functions shall be as follows:

- o SMITE Source Code Analysis
- o MULTI Source Code Analysis
- o Nominal Internal Model Generation
- o Structured Internal Model Generation
- o Flowchart Generation
- o Interactive Response Generation

The SMITE and MULTI Source Code Analysis functions shall each fulfill the requirements described in Section M3.1 for Phase 1, Step 1 of the system. The Nominal Internal Model Generation function shall fulfill the requirements of Phase 1, Step 2. The requirements of Phase 2 shall be fulfilled by the Structured Internal Model Generation function. The Flowchart Generation and Interactive Response Generation functions shall fulfill the requirements specified for Phase 3.

M3.3 Inputs/Outputs

M3.3.1 Inputs

The inputs to the Control Flow Analyzer shall consist of the following:

- o The source code for a microprogram written in SMITE or MULTI
- o User inputs specifying all possible destinations for indirect branches in a MULTI microprogram
- o User-specified flowcharting options
- o User questions about the flow of control within the microprogram

The source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. The three types of user inputs may be entered via card reader or terminal. The user inputs identifying destinations of indirect branches shall make use of the sequence numbers provided in the numbered listing generated by the system. The flowcharting options and user questions shall be as specified in Section M3.1.

M3.3.2 Outputs

The output of the system shall consist of the following:

- o A numbered listing of the microprogram
- o A report on the Phase 1 user inputs specifying destinations of indirect branches
- o A report identifying any unreachable code

- A nominal or structured flowchart of the microprogram

- Responses to user questions about the microprogram flow of control

The format of the numbered listing, the report on Phase 1 user inputs, and the report on unreachable code shall be as indicated in Figures M3, M4, and M5, respectively. These outputs shall be directed to a line printer or to a terminal. The format of the system-generated flowcharts shall be as indicated in Figure M6. These flowcharts may be directed to a line printer, a plotter, or a CRT, depending upon user specification. The exact format will depend upon the output medium selected. Figure M7 illustrates the format of the responses to user questions. These responses shall be directed to a line printer or to a terminal.

M3.4 Data Characteristics

The microprogram to be processed may reside in a system data base. In addition, the system shall create the following files to be stored on disk:

- The tables of control flow information
- The nominal internal model of the microprogram
- The structured internal model of the microprogram

The size of these files will depend upon the microprogram being processed, but the maximum storage required for each file will be approximately 50K words. The first file, produced by the SMITE or MULTI Source Code Analysis function, need be saved only until it is used to generate the internal models. The files containing the internal models are to be saved as long as they may be needed for execution of Phase 3 of the system.

M3.5 Failure Contingencies

Not applicable.

M4. ENVIRONMENT

M4.1 Equipment Environment

The Control Flow Analyzer will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 120K words of main memory to operate. The system will be stored on disk. The microprogram to be analyzed may be stored on punched cards, magnetic tape, or disk. The user inputs may be entered via card reader or terminal. The tables of control flow information and the internal microprogram models shall be stored on disk. The outputs of the system will be directed to a line printer, a terminal, and/or a plotter. No new equipment will be required.

```
STATEMENT #           SOURCE STATEMENT

 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                     .
   .                     .
   .                     .
```

Figure M3. Format of the Numbered Source Listing

```
STATMENT #          DESTINATIONS OF INDIRECT BRANCHES

XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                          .
  .                          .
  .                          .
```

Figure M4. Format of the Report on User Inputs

```
UNREACHABLE SOURCE STATEMENTS

XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

Figure M5. Format of the Report on Unreachable Code

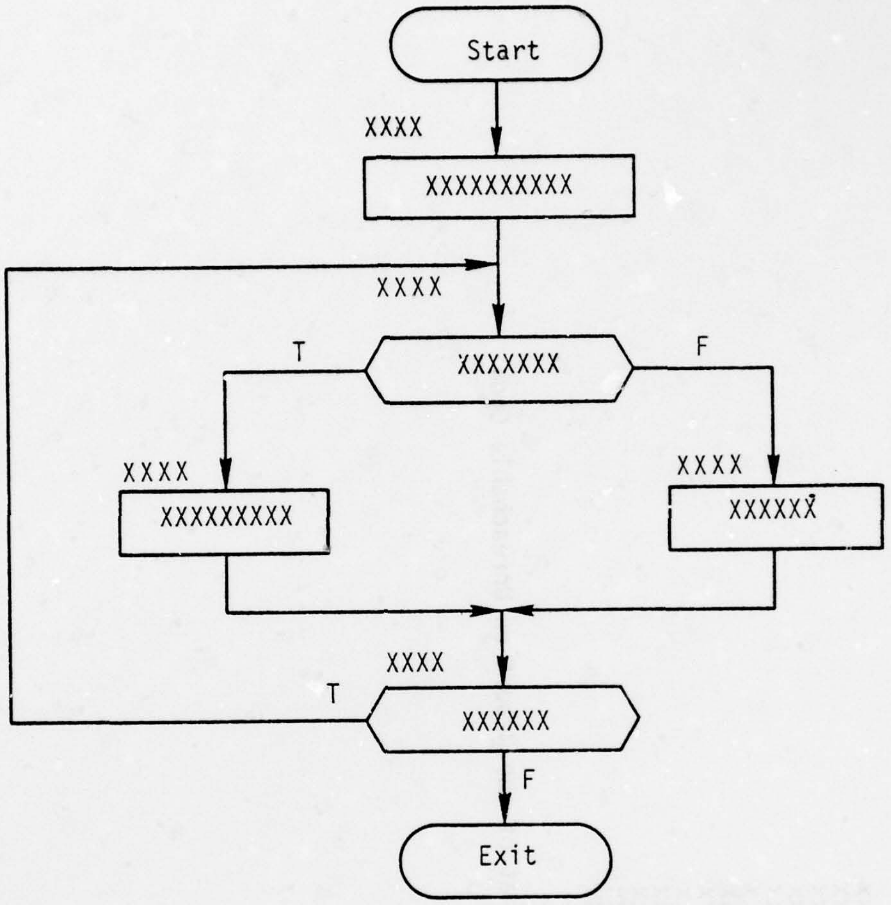
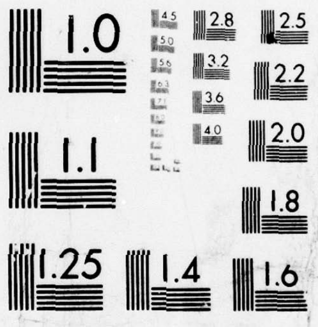


Figure M6. Sample Format of a System-Generated Flowchart

AD-A073 454 LOGICON INC SAN PEDRO CALIF F/G 9/2
RELIABLE MICROPROGRAMMING.(U)
JUL 79 J RADATZ, J LAUB, G HAMACHI F30602-78-C-0079
UNCLASSIFIED DS-R79001 RADC-TR-79-173 NL

3 OF 4

AD A073454


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
SUCCESSORS TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

PREDECESSORS TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

PATHS FROM XXXXX TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

       .

       .

       .
```

Figure M7. Format of the Responses to User Questions

M4.2 Support Software Environment

See Appendix G.

M4.3 Interfaces

The system will not interface with any other system.

M4.4 Security

The system will have no classified components.

M5. COST FACTORS

Development of the Control Flow Analyzer will require approximately 40 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the program.

M6. DEVELOPMENTAL PLAN

The Control Flow Analyzer is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Control Flow Analyzer is given in Figure M8.

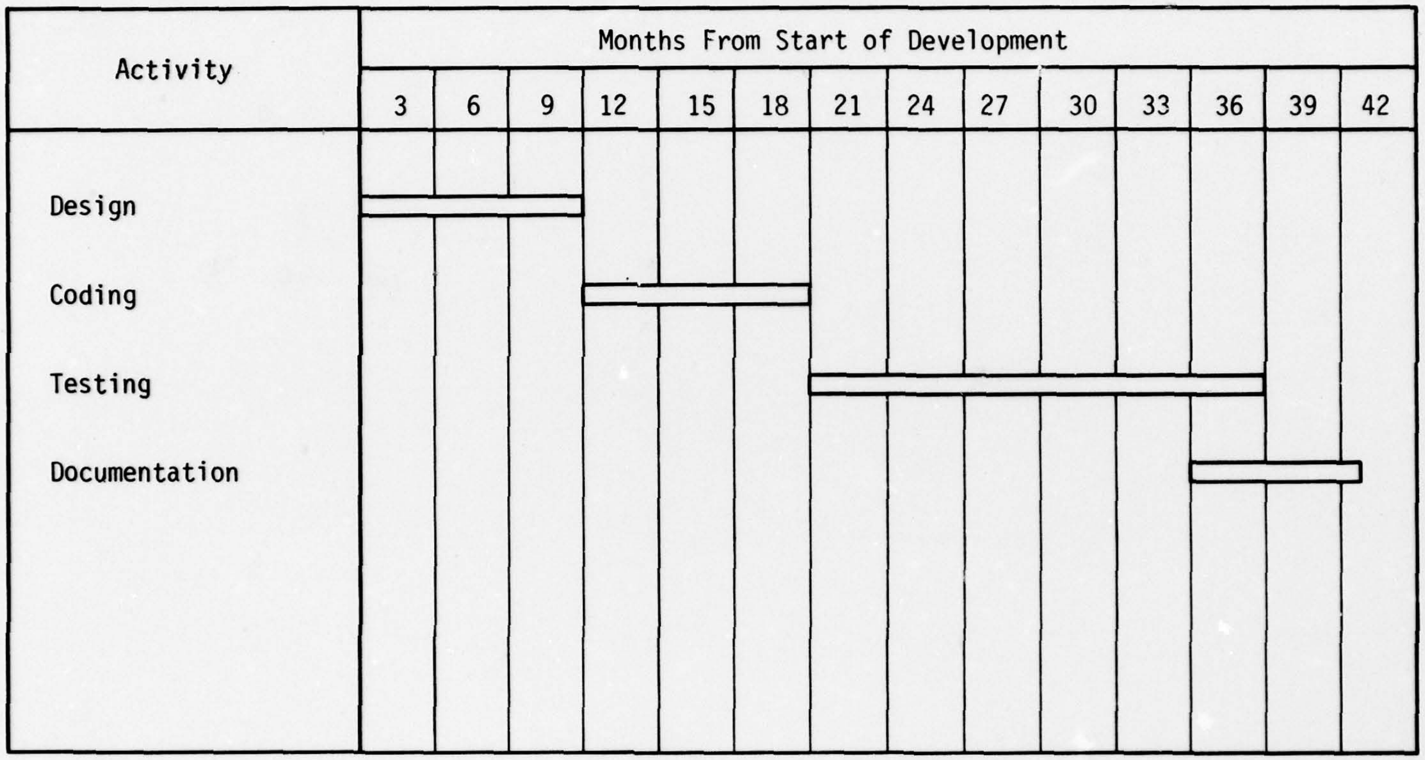


Figure M8. Development Schedule for the Control Flow Analyzer

APPENDIX N
SYSTEM SPECIFICATION FOR THE CONTROL FLOW ANALYZER

N1. GENERAL

N1.1 Purpose of the System Specification

See Appendix H.

N1.2 Project References

See Appendix H.

N2. SUMMARY OF REQUIREMENTS

N2.1 System Description

The Control Flow Analyzer shall provide an automated means of investigating the flow of control within a microprogram written in SMITE or MULTI. The system shall operate in three independent phases. Phase 1 shall analyze the microprogram source code, generate a numbered listing of the microprogram, report on user inputs, build a nominal internal model of the microprogram (i.e., a model of the microprogram as it was written), and produce a report identifying any unreachable code in the microprogram. Phase 2 shall build a structured version of the internal model. Phase 3 shall use the internal models to generate a nominal or structured flowchart of the microprogram and to respond interactively to user questions about the microprogram's flow of control.

The system shall consist of six computer programs. Figure N1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

N2.2 System Functions

The Control Flow Analyzer shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o SMITE Source Code Analysis
- o MULTI Source Code Analysis
- o Nominal Internal Model Generation
- o Structured Internal Model Generation
- o Flowchart Generation
- o Interactive Response Generation

The following paragraphs identify the requirements to be satisfied by each of these functions.

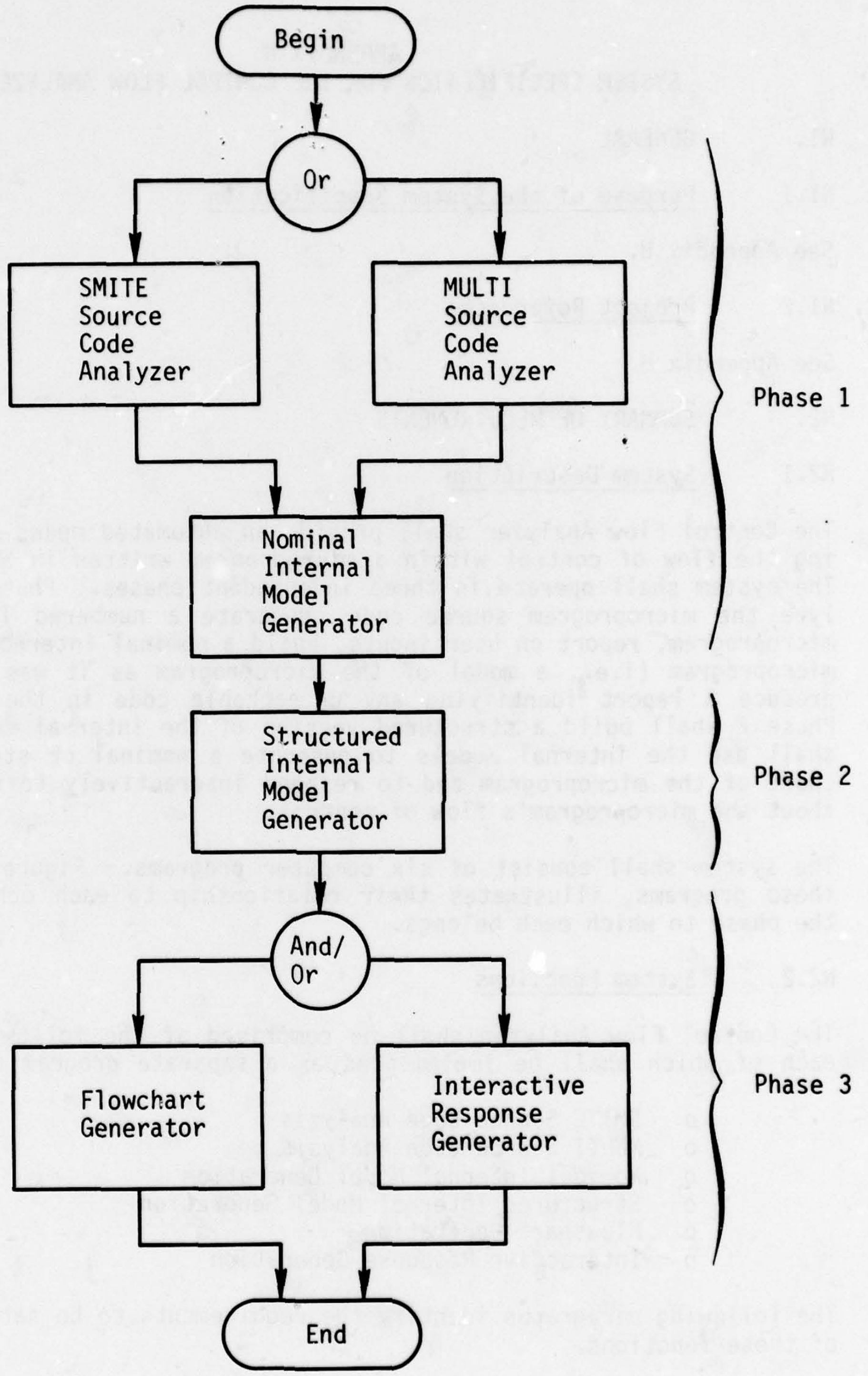

Begin
Or
SMITE Source Code Analyzer
MULTI Source Code Analyzer
Nominal Internal Model Generator
Structured Internal Model Generator
And/ Or
Flowchart Generator
Interactive Response Generator
End
Phase 1
Phase 2
Phase 3

Figure N1. Relationship of the Components of the Control Flow Analyzer

N2.2.1 SMITE Source Code Analysis Function

The SMITE Source Code Analysis function shall accept as input the source code of a program written in SMITE. The function shall process each source statement of the program as follows:

- o Assign the statement a sequence number
- o Output a copy of the statement and its sequence number in a listing of the program
- o Analyze the contents of the statement and store relevant information in control flow tables

Source code comments shall be copied into the program listing without being analyzed or assigned a sequence number. Information to be stored in the control flow tables shall include the following:

- o The location and usage of each label
- o All possible predecessors and successors to each source statement
- o The text of each source statement

To limit execution time and core requirements, the function shall expect the program to be syntactically correct and to have no more than 2,000 variables, 2,000 statements, 500 branches, and 100,000 total characters. If any source statement cannot be interpreted, the function shall report this problem in the program listing and shall proceed to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, then continue to analyze and list the source statements, storing all relevant information for which space remains.

At the conclusion of the source code analysis, the function shall output a problem summary message if any source statements could not be interpreted or if space limitations were exceeded in the processing. The control flow tables shall be stored on disk for use by the Nominal Internal Model Generation function.

N2.2.2 MULTI Source Code Analysis Function

The requirements for the MULTI Source Code Analysis function shall be the same as those given in Section N2.2.1 for the SMITE Source Code Analysis function, with two exceptions:

- o This function shall accept as input the source code of a microprogram written in MULTI.

- o Possible successors to statements containing indirect branches, which cannot be determined from the MULTI source code, shall be left undetermined in the control flow tables.

N2.2.3 Nominal Internal Model Generation Function

The Nominal Internal Model Generation function shall accept as input the control flow tables produced by the SMITE or MULTI Source Code Analysis function and, in the case of a MULTI microprogram, a set of user-prepared inputs specifying all possible destinations for indirect branches in the microprogram. If user inputs are present, the function shall check them for errors in format and content, record the specified branch destinations, and prepare a report that lists each user input and identifies any errors detected. If errors have been detected or if the inputs do not provide all of the required information, the function shall request and process the required corrections and additions. When the inputs are correct and complete, the function shall perform the following tasks:

- o Build a nominal internal model of the microprogram (i.e., a model of the microprogram as it was written)
- o Generate a report identifying any unreachable code

The nominal internal model shall indicate the sequence numbers of all possible predecessors and successors to each source code statement and shall contain the text of each statement for use in flowchart generation. The model shall be stored on disk for use by the Structured Internal Model Generation function, the Flowchart Generation function, and the Interactive Response Generation function.

The report on unreachable code shall identify by sequence number any statement that meets either of the following criteria:

- o It is not a possible successor to any other statement in the microprogram.
- o It is a possible successor only to other unreachable statements.

If there are no such statements, the report shall indicate that the microprogram contains no unreachable code.

N2.2.4 Structured Internal Model Generation Function

The Structured Internal Model Generation function shall accept as input the nominal internal model produced by the Nominal Internal Model Generation function. It shall use this model to build a second model of the microprogram, one in which the flow of control is expressed in terms of the structured programming constructs of sequential code, binary branches, and single-entry, single-exit loops. The model shall indicate the flow of control by identifying the sequence numbers of all possible predecessors and successors to each source statement and shall contain the text of each source statement for use in flowchart generation. The model shall be stored on disk for use by the Flowchart Generation and Interactive Response Generation functions.

N2.2.5 Flowchart Generation Function

The Flowchart Generation function shall be responsible for producing a microprogram flowchart. Inputs to the function shall be as follows:

- o A set of user-prepared inputs specifying flowcharting options
- o The nominal or structured internal model of a microprogram, depending upon the type of flowchart desired

The flowcharting options that may be specified shall be as follows:

- o Use of ANSI Standard flowchart symbols
- o Minimum border width
- o Character height
- o Collapsing of consecutive process boxes
- o Production of a directory of connector occurrences
- o Appending of labels to the flowchart
- o Substitution of mathematical notation for logical and relational operators
- o Use of statement identifiers in place of source text
- o Output device (printer, plotter, or CRT)
- o Page height and width
- o Output of source prologue
- o Segmentation of the flowchart
- o Production of a structured flowchart from unstructured input

Each of these specifications shall be optional. The function shall check the inputs for errors in format and content, report any errors, and record the options that are to be used, providing default values for any options not specified by the user. The default values shall be as follows:

- o Use of non-ANSI symbols
- o Line printer or CRT borders of 0 characters; plotter borders of 1.0 inch at the top, bottom, and right, and 1.25 inches at the left
- o Plotter character height of 0.125 inch
- o Collapsing of consecutive process boxes
- o No directory of connector occurrences

- o Appending of labels to the flowchart
- o Substitution of mathematical notation for logical and relational operators
- o Use of source text in the flowchart
- o Directing of output to a line printer
- o Line-printer page dimensions of 999 lines by 131 characters; CRT and plotter page dimensions of 11.0 inches by 8.5 inches
- o No output of a prologue
- o No flowchart segmentation
- o Generation of a structured flowchart from unstructured input

If errors are detected in the inputs, the function shall request and process the necessary corrections. When the inputs are correct, the function shall generate the desired flowchart. It shall be a detailed, source-level flowchart incorporating the user-specified options.

N2.2.6 Interactive Response Generation Function

The Interactive Response Generation function shall accept as input the nominal and/or structured internal models, together with user questions regarding the microprogram flow of control. The following types of questions shall be handled:

- o What statements might immediately follow statement N during microprogram execution?
- o What statements might immediately precede statement N during microprogram execution?
- o What are all possible paths from statement M to statement N?

For questions of the first two types, the function shall be able to use either internal model; for questions of the third type, it shall use the structured model. Questions and responses shall identify source statements by the sequence numbers assigned to them in the numbered listing produced during source code analysis. If a user question cannot be interpreted, the function shall permit the user to correct the question. In enumerating microprogram paths, the function shall limit itself to a single execution of each loop.

N2.3 <u>Accuracy and Validity</u>

Accuracy and validity requirements are not applicable to the system.

N2.4 <u>Timing</u>

There are no timing requirements on the system.

N2.5 <u>Flexibility</u>

The system shall be designed in such a way that the limit on the number of variables, statements, branches, and characters can be easily modified.

N3. ENVIRONMENT

N3.1 <u>Equipment Environment</u>

See Appendix H for equipment description.

The Control Flow Analyzer will require approximately 120K words of main memory. The system will be stored on disk, requiring approximately 60K words of storage for the source code, object code, and load module. User-prepared inputs may be entered via card reader or terminal. Microprogram source may be stored on punched cards, magnetic tape, or disk. System-generated files will be stored on disk. Output will be directed to the line printer, to a plotter, and/or to a terminal. No new equipment will be required.

N3.2 <u>Support Software Environment</u>

See Appendix H.

N3.3 <u>Interfaces</u>

The system shall not interface with any other system.

N3.4 <u>Security</u>

The system shall have no classified components.

N3.5 <u>Controls</u>

No controls shall be established by the system.

N4. DESIGN DATA

N4.1 <u>System Logical Flow</u>

- o The SMITE Source Code Analyzer
- o The MULTI Source Code Analyzer

- o The Nominal Internal Model Generator
- o The Structured Internal Model Generator
- o The Flowchart Generator
- o The Interactive Response Generator

The first three of these programs shall constitute Phase 1 of the system. To execute this phase, the user would invoke either the SMITE or MULTI Source Code Analyzer, then invoke the Nominal Internal Model Generator. The next program comprises Phase 2, execution of which is optional. The last two programs comprise Phase 3 of the system. To execute this phase, the user would invoke one or both programs. If both are invoked, their order of execution is immaterial. For a given microprogram version, Phase 1 is meant to be executed once, Phase 2 once or not at all, and Phase 3 any number of times. Figure N2 indicates the logical flow of the system for a case in which all three phases are executed and in which the Flowchart Generator is invoked before the Interactive Response Generator. The figure also indicates the inputs and outputs of each program, including the data base files created and used by each program.

N4.2 Program Descriptions

The Control Flow Analyzer shall be comprised of the six computer programs identified in Section N4.1. Design data for each program follow.

N4.2.1 SMITE Source Code Analyzer Program

The SMITE Source Code Analyzer shall fulfill the requirements specified in Section N2.2.1. Additional design data follow.

N4.2.1.1 Inputs: The input to the program shall consist of the source code of a program written in SMITE. This source code shall be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

N4.2.1.2 Outputs: The output of the program shall consist of control flow tables, described in Section N4.2.1.3, and a numbered listing of the SMITE program. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure N3. If problems are encountered, they shall be identified in the listing and summarized in a message at the conclusion of the listing.

N4.2.1.3 Data Base: The SMITE program to be processed may reside in a system data base. In addition, the program shall store on disk control flow tables for use by the Nominal Internal Model Generator. These tables shall indicate the location and usage of each program label, all possible predecessors and successors to each source statement, and the text of each

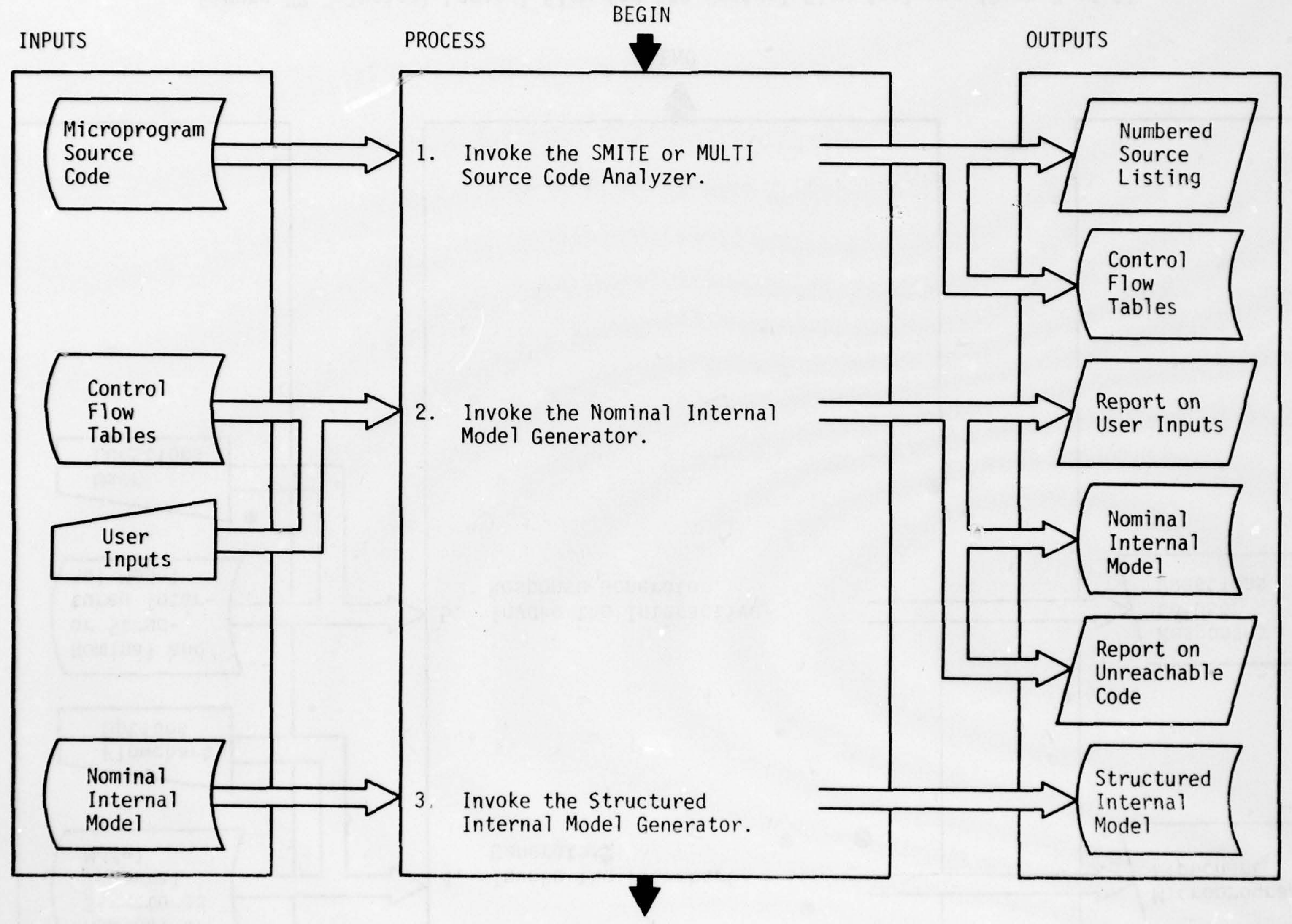

BEGIN
INPUTS
PROCESS
OUTPUTS
Microprogram Source Code
Control Flow Tables
User Inputs
Nominal Internal Model
1. Invoke the SMITE or MULTI Source Code Analyzer.
2. Invoke the Nominal Internal Model Generator.
3. Invoke the Structured Internal Model Generator.
Numbered Source Listing
Control Flow Tables
Report on User Inputs
Nominal Internal Model
Report on Unreachable Code
Structured Internal Model

Figure N2. Typical Logical Flow for the Control Flow Analyzer (Page 1 of 2)

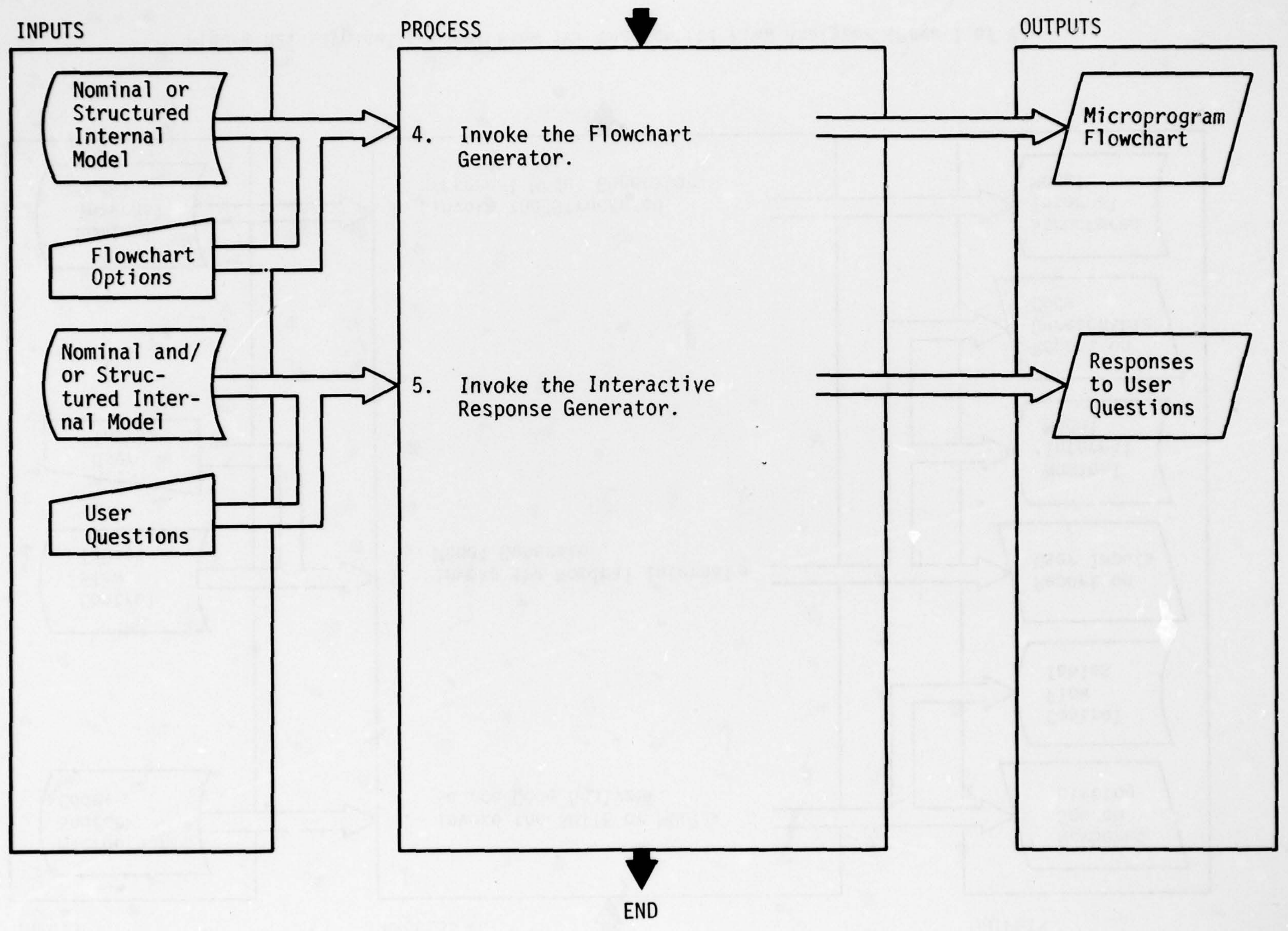

INPUTS
PROCESS
OUTPUTS
Nominal or Structured Internal Model
Flowchart Options
Nominal and/or Structured Internal Model
User Questions
4. Invoke the Flowchart Generator.
5. Invoke the Interactive Response Generator.
Microprogram Flowchart
Responses to User Questions
END

Figure N2. Typical Logical Flow for the Control Flow Analyzer (Page 2 of 2)

```
STATEMENT #           SOURCE STATEMENT

  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    .                      .
    .                      .
    .                      .
```

Figure N3. Format of the Numbered Source Listing

source statement. The format of the tables shall be determined during detailed design of the system. The size of the file will depend upon the SMITE program being analyzed, but should not exceed 50K words. The file need be retained only until it is used by the Nominal Internal Model Generator.

N4.2.2 MULTI Source Code Analyzer Program

The MULTI Source Code Analyzer shall fulfill the requirements specified in Section N2.2.2. Additional design data follow.

N4.2.2.1 Inputs: The input to the program shall consist of the source code of a microprogram written in MULTI. This source code shall be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the Nanodata Corporation report "MULTI Micromachine Description" (2 February 1976).

N4.2.2.2 Outputs: The output of the program shall consist of control flow tables, described in Section N4.2.2.3, and a numbered listing of the microprogram. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure N3. If problems are encountered, they shall be identified in the listing and summarized in a message at the conclusion of the listing.

N4.2.2.3 Data Base: The microprogram to be processed may reside in a system data base. In addition, the program shall store on disk control flow tables for use by the Nominal Internal Model Generator. These tables shall indicate the location and usage of each microprogram label, all possible predecessors and successors to each source statement except indirect branches, and the text of each source statement. The format of this file shall be determined during detailed design of the system. The size of the file will depend upon the microprogram being analyzed, but should not exceed 50K words. The file need be retained only until it is used by the Nominal Internal Model Generator.

N4.2.3 Nominal Internal Model Generator Program

The Nominal Internal Model Generator shall fulfill the requirements specified in Section N2.2.3. Additional design data follow.

N4.2.3.1 Inputs: For a SMITE program, the inputs to this program shall be the control flow tables described in Section N4.2.1.3. For a MULTI microprogram, the inputs shall consist of the control flow tables described in Section N4.2.2.3 plus user-prepared inputs specifying all possible destinations for each indirect branch in the microprogram. The user input shall be entered via card reader or terminal. Its format shall be determined during detailed design of the program.

N4.2.3.2 Outputs: The output of the program shall consist of the following:

- o A report on user inputs
- o A report identifying any unreachable code in the microprogram
- o A nominal internal model of the microprogram

Both of the reports shall be directed to a line printer or to a terminal. If the user inputs are correct and complete, the report on them shall have the format indicated in Figure N4. The report on unreachable code shall have the format indicated in Figure N5. If there is no unreachable code in the microprogram, the list of sequence numbers shall be replaced by a message stating that no unreachable code exists. The nominal internal model is described in Section N4.2.3.3.

N4.2.3.3 Data Base: The program shall make use of the tables of control flow information stored on disk by the SMITE or MULTI Source Code Analyzer. These tables are described in Sections N4.2.1.3 and N4.2.2.3. In addition, the program shall store on disk a nominal internal model of the microprogram for use by the Structured Internal Model Generator, the Flowchart Generator, and/or the Interactive Response Generator. This model shall indicate the sequence numbers of all possible predecessors and successors to each source statement and shall contain the text of each source statement. Its format shall be determined during detailed design of the system. Its size will depend upon the microprogram being analyzed, but should not exceed 50K words. The model is to be retained as long as it may be needed for executions of Phases 2 and 3 of the system.

N4.2.4 Structured Internal Model Generator Program

The Structured Internal Model Generator shall fulfill the requirements specified in Section N2.2.4. Additional design data follow.

N4.2.4.1 Inputs: The input to the program shall be the nominal internal model described in Section N4.2.3.3.

N4.2.4.2 Outputs: The output from the program shall be the structured internal model described in Section M4.2.4.3.

N4.2.4.3 Data Base: The program shall make use of the nominal internal model stored on disk by the Nominal Internal Model Generator. This model is described in Section N4.2.3.3. The program shall store on disk a structured internal model of the microprogram for use by the Flowchart Generator and/or the Interactive Response Generator. This model shall express the microprogram's flow of control in terms of structured constructs. It shall indicate all possible predecessors and successors to each source statement and shall contain the text of each source statement. Its format shall be determined during detailed design of the system. Its

```
STATMENT #        DESTINATIONS OF INDIRECT BRANCHES

XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                        .
  .                        .
  .                        .
```

Figure N4. Format of the Report on User Inputs

```
UNREACHABLE SOURCE STATEMENTS

XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

Figure N5. Format of Report on Unreachable Code

size will depend upon the microprogram being analyzed, but should not exceed 50K words. The model is to be retained as long as it may be needed for executions of Phase 3 of the system.

N4.2.5 Flowchart Generator Program

The Flowchart Generator shall fulfill the requirements specified in Section N2.2.5. Additional design data follow.

N4.2.5.1 Inputs: The inputs to the program shall consist of the nominal or structured internal model of a microprogram, and a set of user-prepared inputs specifying flowcharting options. The nominal and structured internal models are described in Sections N4.2.3.3 and N4.2.4.3, respectively. The flowcharting options shall be as follows:

- Whether the flowcharts should use ANSI Standard flowchart symbols or space-saving symbols consisting of truncated diamonds for tests and rectangles of unrestricted proportions for processing
- The minimum width of the border between the flowchart and each edge of the page
- The height of the text characters to be used in plotter-output flowcharts
- Whether consecutive process boxes are to be collapsed into a single box or drawn separately
- Whether a directory summarizing all connector occurrences is to be printed after the flowchart
- Whether the flowchart boxes are to have source code labels appended to them
- Whether logical and relational operators are to be replaced by mathematical symbols in the text of tests
- Whether the source code text is to be replaced by statement identifiers from the source listing
- Whether the flowchart is to be output on a line printer, a plotter, or a CRT
- The horizontal and vertical size of each flowchart page
- Whether a prologue, consisting of a block of comments preceding the source code, is to be printed

- o Whether a segmented flowchart is to be drawn, i.e., whether code segments that can be conveniently drawn separately are to become independent subflows referenced by the main flow
- o Whether a structured flowchart is to be produced from unstructured input

These inputs shall be entered via card reader or terminal. Their format shall be determined during detailed design of the system.

N4.2.5.2 Outputs: The output of the program shall consist of a nominal or structured flowchart of the microprogram. The flowchart shall be directed to a line printer, a plotter, or a CRT, depending upon user request. Figure N6 indicates the format of a typical flowchart directed to a plotter.

N4.2.5.3 Data Base: The program shall make use of the nominal internal model stored on disk by the Nominal Internal Model Generator or the structured internal model stored on disk by the Structured Internal Model Generator. The former is described in Section N4.2.3.3, the latter in Section N4.2.4.3. The file used shall be determined by the user based on the type of flowchart desired.

N4.2.6 Interactive Response Generator Program

The Interactive Response Generator shall fulfill the requirements specified in Section N2.2.6. Additional design data follow.

N4.2.6.1 Inputs: The inputs to the program shall consist of the nominal and/or the structured internal model, and a set of user questions about the microprogram's flow of control. The nominal and structured internal models are described in Sections N4.2.3.3 and N4.2.4.3, respectively. The types of questions that may be input shall be as specified in Section N2.2.6. The format of these questions shall be determined during detailed design of the system. The questions shall be input via card reader or terminal.

N4.2.6.2 Outputs: The output of the program shall consist of responses to the user questions. These responses shall be directed to a line printer or to a terminal. The format of the output shall be as indicated in Figure N7. If any user question cannot be interpreted, an error message shall be output in place of a response, enabling the user to correct and resubmit the question.

N4.2.6.3 Data Base: The program shall make use of the nominal internal model stored on disk by the Nominal Internal Model Generator and/or the structured internal model stored on disk by the Structured Internal Model Generator. The nominal model is described in Section N4.2.3.3, the structured model in Section N4.2.4.3.

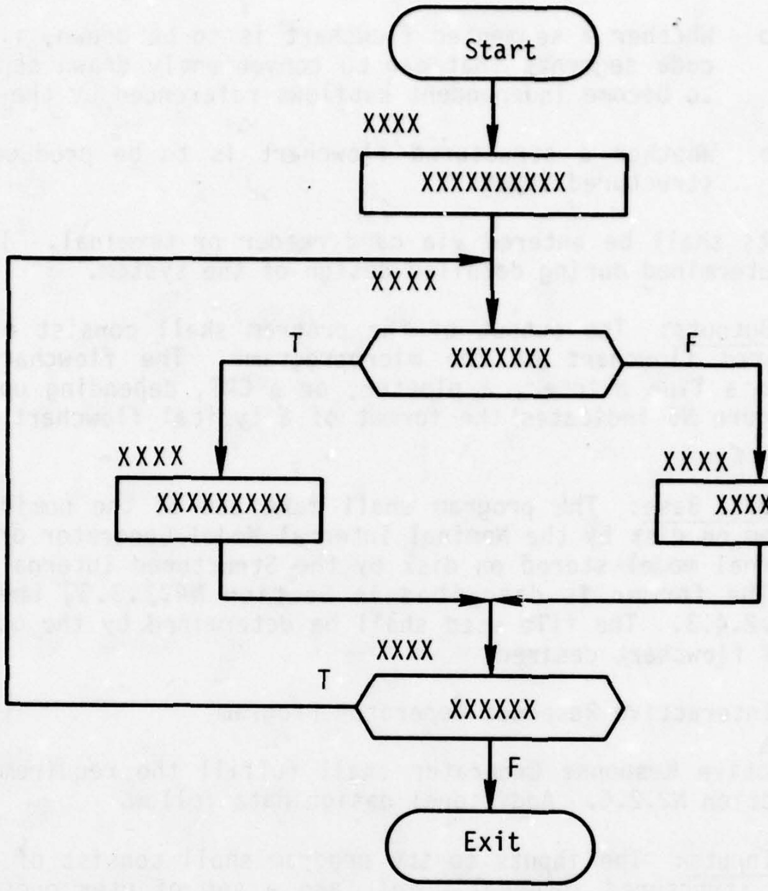


Figure N6. Sample Format of a System-Generated Flowchart

```
SUCCESSORS TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

PREDECESSORS TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

PATHS FROM XXXXX TO XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

     XXXXX
     XXXXX
       .
       .
       .
     XXXXX

       .

       .

       .
```

Figure N7. Format of the Responses to User Questions

APPENDIX O
FUNCTIONAL DESCRIPTION FOR THE TIMING ANALYZER

O1. GENERAL

O1.1 Purpose of Functional Description

See Appendix G.

O1.2 Project References

See Appendix G.

O2. SYSTEM SUMMARY

O2.1 Background

See Appendix G.

O2.2 Objectives

The purpose of the Timing Analyzer is to provide an automated means of determining the execution time of selected portions of a microprogram written in MULTI. Using a table of timing formulas, the system will be capable of responding interactively to user questions about the time required to execute specified microprogram paths.

O2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of a Timing Analyzer. Determination of the amount of time required to execute a given microprogram path is a manual process that involves looking up the execution time of each microinstruction of the path, performing calculations for microinstructions that have variable execution times, and summing the results.

O2.4 Proposed Methods and Procedures

The proposed method for investigating the execution time of selected paths of a microprogram is to submit the microprogram to the Timing Analyzer. The system will operate in two phases. Phase 1 will analyze the microprogram source code, produce a numbered listing of the microprogram, process and report on user inputs specifying indirect branch destinations, and generate an internal model of the microprogram. Phase 2 will use the internal model and a control file of timing formulas to respond interactively to user questions about the execution time of selected microprogram paths. Figure O1 illustrates the data flow for the system. Figure O2 shows its major processing steps.

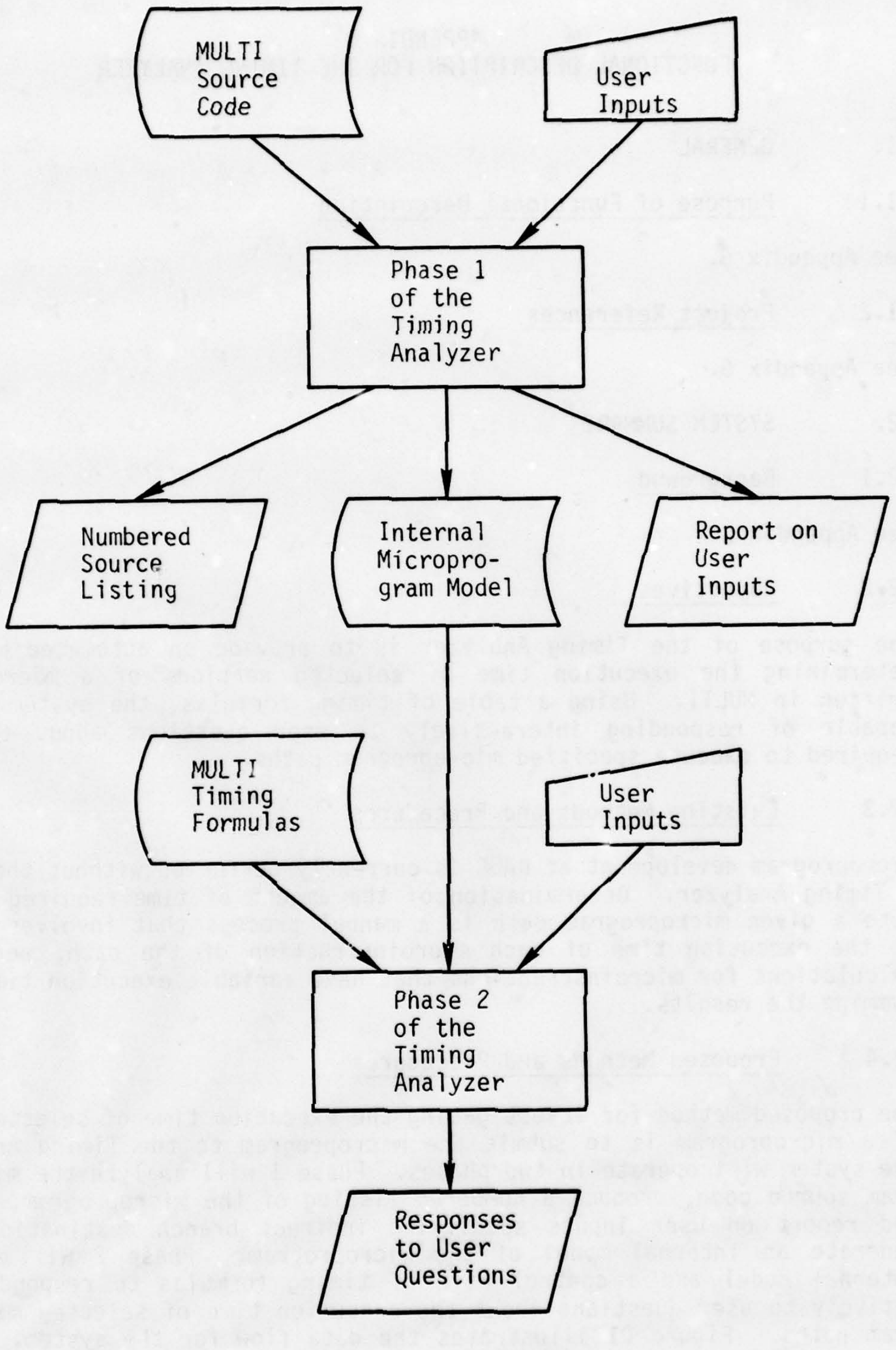

MULTI Source Code
User Inputs
Phase 1 of the Timing Analyzer
Numbered Source Listing
Internal Micropro-gram Model
Report on User Inputs
MULTI Timing Formulas
User Inputs
Phase 2 of the Timing Analyzer
Responses to User Questions

Figure 01. Data Flow for the Timing Analyzer

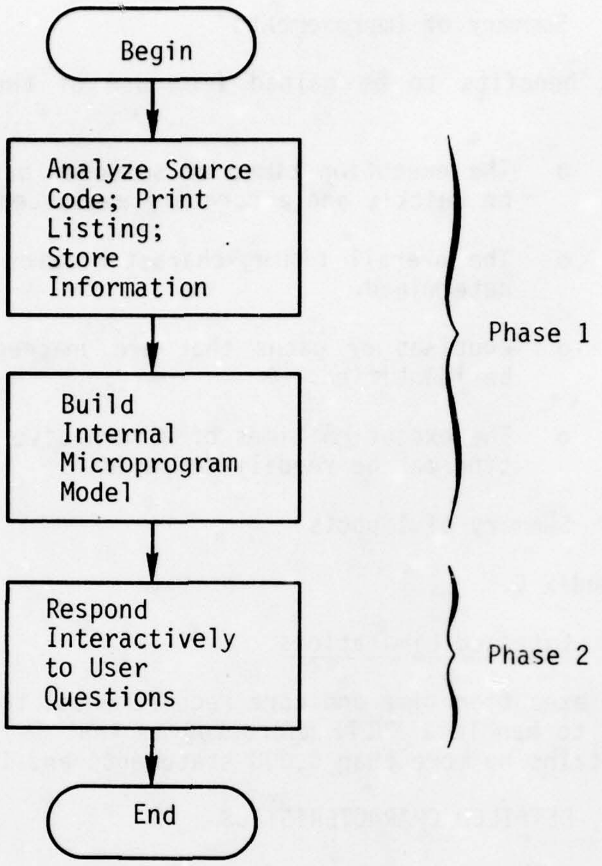

Begin
Analyze Source Code; Print Listing; Store Information
Build Internal Microprogram Model
Phase 1
Respond Interactively to User Questions
Phase 2
End

Figure 02. Major Processing Steps in the Timing Analyzer

02.4.1 Summary of Improvements

Specific benefits to be gained from use of the Timing Analyzer are as follows:

- o The execution times of selected paths of a microprogram can be quickly and accurately evaluated and compared.
- o The overall timing characteristics of a microprogram can be determined.
- o Routines or paths that are unacceptably time consuming can be identified.
- o The execution times of alternative implementations of a routine can be readily compared.

02.4.2 Summary of Impacts

See Appendix G.

02.5 <u>Expected Limitations</u>

To limit execution time and core requirements, the Timing Analyzer will be designed to handle a MULTI microprogram that is syntactically correct and that contains no more than 5,000 statements and 1,000 branches.

03. DETAILED CHARACTERISTICS

03.1 <u>Specific Performance Requirements</u>

The Timing Analyzer shall operate in two phases, each designed to be executed independently. Phase 1 shall be responsible for performing the following tasks:

- o Analyzing the source code of a microprogram written in MULTI
- o Assigning each source statement a sequence number and generating a numbered listing of the microprogram
- o Processing and reporting on user inputs, if any
- o Building an internal model of the microprogram

Phase 1 shall operate in two steps. In the first step, the source code shall be analyzed, the numbered listing shall be generated, and control flow and timing information shall be stored in control flow tables. If any source statement cannot be interpreted during this step, it shall be identified in the numbered listing and the system shall proceed to the next source statement. If any of the numerical limitations given in Section 02.5 is exceeded during this processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

The second step of Phase 1 shall consist of building an internal model of the microprogram, using the information stored in tables during Step 1 and a set of user inputs specifying all possible destinations for each indirect branch in the microprogram. The user inputs shall make use of the sequence numbers provided in the numbered listing generated in Step 1. The system shall process the user inputs, check them for errors in format and content, and prepare a report on them, identifying any errors detected. If errors are present, the system shall request and process corrections. When the inputs are correct, the system shall use them to build an internal model of the microprogram. The internal model shall identify all possible successors to each source statement and shall contain the syntactical information needed to determine each statement's execution time. The internal model shall be stored on disk to be used as input to Phase 2.

In Phase 2, the system shall use the internal model, together with a control file of timing formulas, to respond interactively to user questions about the time required to execute specified paths through the microprogram. A user question shall specify the first and last source statements on the desired path, the desired outcome of each branch along the path, and the desired number of iterations of each loop on the path. Any information not provided in the initial question shall be requested by the system during the timing analysis. The user question shall also specify the desired handling of microinstructions that have variable execution times. The user shall be able to select any one or more of the following options:

- o Calculate the path's minimum execution time by using the minimum execution time for each variably timed microinstruction

- o Calculate the path's maximum execution time by using the maximum execution time for each variably timed microinstruction

- o Calculate a representative execution time for the path by using a representative execution time for each variably timed microinstruction

- o Calculate a specific execution time for the path by employing additional user inputs to compute an exact execution time for each variably timed microinstruction

If a user question cannot be interpreted or contains an error, the system shall report the problem to the user and request correction. The timing analysis shall be performed using the internal model, the control file of timing formulas for MULTI microinstructions, and the user specifications. If additional user information is required at any point in the processing, the system shall request the additional information, process the user response, then proceed. When the final microinstruction on a path has been processed, the system shall output a path description identifying the sequence of statements encountered on the path and shall report the total execution time(s) for the path. The system shall then permit the user to submit another question or to terminate the timing analysis.

03.1.1 Accuracy and Validity

The accuracy of the reported execution times will depend upon the consistency of the execution times of the instruction set being analyzed.

03.1.2 Timing

There are no timing requirements on the system.

03.2 System Functions

The Timing Analyzer shall perform three major functions, each of which shall be implemented as a separate program in the system. These functions shall be as follows:

- o MULTI Source Code Analysis
- o Internal Model Generation
- o Interactive Response Generation

The MULTI Source Code Analysis function shall fulfill the requirements described in Section 03.1 for Phase 1, Step 1 of the system. The Internal Model Generation function shall fulfill the requirements of Phase 1, Step 2. The requirements of Phase 2 shall be fulfilled by the Interactive Response Generation function.

03.3 Inputs/Outputs

03.3.1 Inputs

The inputs to the Timing Analyzer shall consist of the following:

- o The source code for a microprogram written in MULTI
- o User inputs specifying all possible destinations for the indirect branches in the microprogram
- o User questions about the time required to execute specified paths in the microprogram

The source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. The user inputs and user questions may be entered via card reader or terminal. The format of these inputs shall be determined during detailed design of the system.

03.3.2 Outputs

The output of the system shall consist of the following:

- o A numbered listing of the microprogram
- o A report on the Phase 1 user inputs specifying destinations of indirect branches
- o Responses to user questions about the execution times of selected microprogram paths

The formats of these outputs shall be as indicated in Figures 03, 04, and 05, respectively. Each output shall be directed to a file that may be output on a line printer or a terminal.

03.4 Data Characteristics

The microprogram to be processed may reside in a system data base. In addition, the system shall create the following files to be stored on disk:

- o A file containing the control flow tables stored during source code analysis
- o A file containing the internal model of the microprogram

The size of these files will depend upon the microprogram being processed, but should not exceed 50K words each. The first file need be retained only until it is used to create the internal model. The second file is to be saved as long as it may be needed for executions of Phase 2 of the system.

Phase 2 will also make use of a control file containing timing formulas for MULTI microinstructions. This file will be created as part of the Timing Analyzer system. It will be stored on disk, requiring approximately 5,000 words, and is to be retained as long as the Timing Analyzer system is being used.

03.5 Failure Contingencies

Not applicable.

04. ENVIRONMENT

04.1 Equipment Environment

The Timing Analyzer will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 100K words of main memory to operate. The system will be stored on disk. The microprogram to be analyzed may be stored on punched cards, magnetic tape, or disk. The user

```
STATEMENT #           SOURCE STATEMENT

   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     .                    .
     .                    .
     .                    .
```

Figure 03. Format of the Numbered Source Listing

```
STATMENT #        DESTINATIONS OF INDIRECT BRANCHES

XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                        .
  .                        .
  .                        .
```

Figure 04. Format of the Report on User Inputs

```
MICROPROGRAM PATH

  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

MIN TIME   XXXX

MAX TIME   XXXX

REPRESENTATIVE TIME   XXXX

SPECIFIC TIME   XXXX
```

Figure 05. Format of the Responses to User Questions

inputs may be entered via card reader or terminal. The tables of control flow information, the internal model, and the control file of timing formulas will be stored on disk. The outputs of the system will be directed to a file that may be output on a line printer or a terminal. No new equipment will be required.

04.2 Support Software Environment

See Appendix G.

04.3 Interfaces

The system will not interface with any other system.

04.4 Security

The system will have no classified components.

05. COST FACTORS

Development of the Timing Analyzer will require approximately 10 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the program.

06. DEVELOPMENTAL PLAN

The Timing Analyzer is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Timing Analyzer is given in Figure 06.

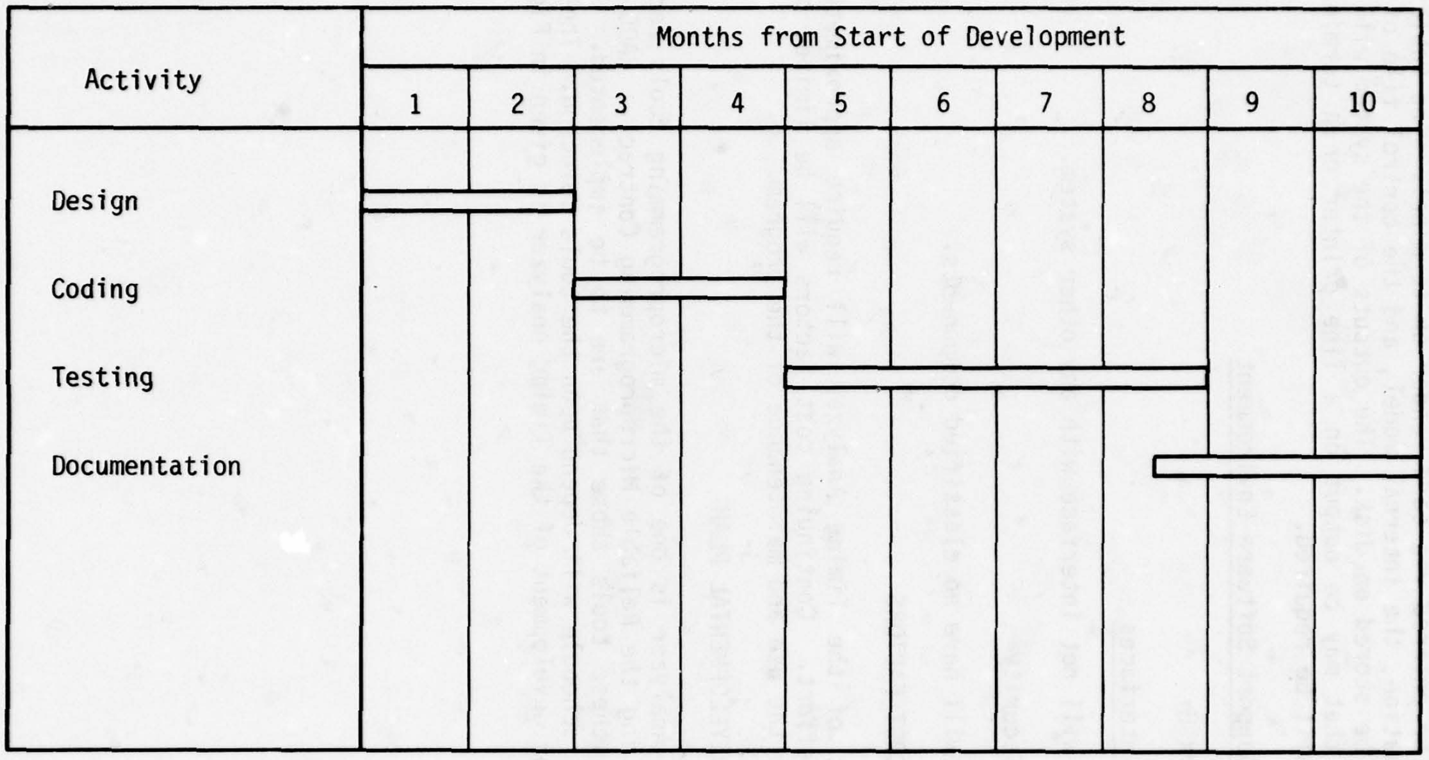


Figure 06. Development Schedule for the Timing Analyzer

APPENDIX P
SYSTEM SPECIFICATION FOR THE TIMING ANALYZER

P1. GENERAL

P1.1 Purpose of the System Specification

See Appendix H.

P1.2 Project References

See Appendix H.

P2. SUMMARY OF REQUIREMENTS

P2.1 System Description

The Timing Analyzer shall provide an automated means of determining the execution time of selected paths of a microprogram written in MULTI. The system shall operate in two independent phases. Phase 1 shall analyze the microprogram source code, produce a numbered listing of the microprogram, process and report on user inputs specifying indirect branch destinations, and generate an internal model of the microprogram. Phase 2 shall use the internal model and a control file of timing formulas to respond interactively to user questions about the execution time of selected microprogram paths.

The system shall consist of three computer programs. Figure P1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

P2.2 System Functions

The Timing Analyzer shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o MULTI Source Code Analysis
- o Internal Model Generation
- o Interactive Response Generation

The following paragraphs identify the requirements to be satisfied by each of these functions.

P2.2.1 MULTI Source Code Analysis Function

The MULTI Source Code Analysis function shall accept as input the source code of a microprogram written in MULTI. The function shall process each source statement of the microprogram as follows:

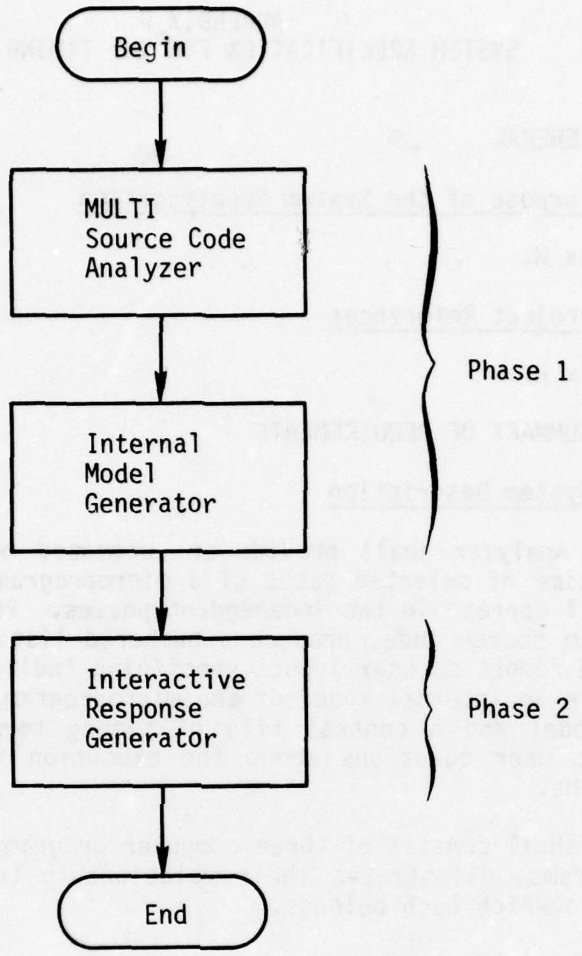


Figure P1. Relationship of the Components of the Timing Analyzer

- Assign the statement a sequence number
- Output a copy of the statement and its sequence number in a listing of the microprogram
- Analyze the contents of the statement and store relevant information in control flow tables

Source code comments shall be copied into the microprogram listing without being analyzed or assigned a sequence number. Information to be stored in the control flow tables shall include the following:

- The location and usage of each label
- All possible successors to each source statement except those containing indirect branches
- Syntactical information needed to determine the execution time of each statement

To limit execution time and core requirements, the function shall expect the microprogram to be syntactically correct and to have no more than 5,000 statements and 1,000 branches. If any source statement cannot be interpreted, the function shall report this problem in the numbered listing and shall proceed to the next source statement. If either of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, then continue to analyze and list each source statement, storing all relevant information for which space remains.

At the conclusion of the source code analysis, the function shall output a problem summary message if any source statements could not be interpreted or if space limitations were exceeded during the processing. The control flow tables shall be stored on disk for use by the Internal Model Generation function.

P2.2.2 Internal Model Generation Function

The Internal Model Generation function shall accept as input the control flow tables produced by the MULTI Source Code Analysis function and a set of user-prepared inputs specifying all possible destinations for each indirect branch in the microprogram. The function shall check the user-prepared inputs for errors in format and content, record the specified branch destinations, and prepare a report that lists each user input and identifies any errors detected. If errors have been detected or if the inputs do not provide all of the required information, the function shall request and process corrections and additions. When the inputs are correct and complete, the function shall proceed to generate an internal model of the microprogram. This model shall indicate for each source statement the sequence number of all possible successors and the syntactical information needed to determine the statement's execution time. The model shall be stored on disk for use by the Interactive Response Generation function.

P2.2.3 Interactive Response Generation Function

The Interactive Response Generation function shall accept as input the internal microprogram model produced by the Internal Model Generation function, a control file of timing formulas for MULTI microinstructions, and a set of user-prepared questions concerning the execution time of selected paths of the microprogram. Each user question shall specify the following information:

- The sequence numbers of the first and last source statements on the desired path
- The desired outcome of each branch along the path
- The desired number of iterations of each loop on the path
- The desired handling of microinstructions that have variable execution times

The function shall permit the user to select any one or more of the following methods of handling the variably timed microinstructions.

- Calculate the path's minimum execution time by using the minimum execution time for each variably timed microinstruction
- Calculate the path's maximum execution time by using the maximum execution time for each variably timed microinstruction
- Calculate a representative execution time for the path by using a representative execution time for each variably timed microinstruction
- Calculate a specific execution time for the path by employing additional user inputs to compute an exact execution time for each variably timed microinstruction

The function shall permit the user to submit a complete question before processing begins or an initial question specifying any subset of the required information. In the latter case, the function shall request additional information as it is required during the timing analysis. If a user question cannot be interpreted or contains an error, the function shall report the problem to the user and request corrections. When a correct question has been submitted, the function shall perform the timing analysis using the internal model, the control file of timing formulas for MULTI microinstructions, and the user specifications. To perform this analysis, the function shall proceed along the specified path, determine the appropriate execution time(s) for each statement on the path, and maintain each type of total that has been requested. If additional user information is required at any point in the processing, the function shall request the additional information, process the user response, then proceed. When the final microinstruction on the path has been processed, the function shall output a path description identifying the sequence of statements encountered on the path and shall report the total execution time(s) for the path. The function shall then permit the user to submit another question or to terminate the timing analysis.

P2.3 Accuracy and Validity

The accuracy of the reported execution times will depend upon the consistency of the execution times of the instruction set being analyzed.

P2.4 Timing

There are no timing requirements on the system.

P2.5 Flexibility

The system shall be designed in such a way that the limit on the number of statements and branches can be easily modified.

P3. ENVIRONMENT

P3.1 Equipment Environment

See Appendix H for equipment description.

The Timing Analyzer will require approximately 100K words of main memory. The program will be stored on disk, requiring approximately 30K words of storage for the source code, object code, and load module. The user-prepared inputs may be entered via card reader or terminal. The microprogram source code may be input from punched cards, magnetic tape, or disk. The system-generated files shall be stored on disk, requiring a maximum of 50K words of storage each. The control file of timing formulas shall be stored on disk, requiring approximately 5,000 words of storage. The outputs may be directed to a line printer or to a terminal. No new equipment will be required.

P3.2 Support Software Environment

See Appendix H.

P3.3 Interfaces

The system shall not interface with any other system.

P3.4 Security

The system shall have no classified components.

P3.5 Controls

No controls shall be established by the system.

P4. DESIGN DATA

P4.1 System Logical Flow

The Timing Analyzer shall be a system consisting of three computer programs:

- o The MULTI Source Code Analyzer
- o The Internal Model Generator
- o The Interactive Response Generator

The first two of these programs shall constitute Phase 1 of the system. The third shall constitute Phase 2. For any given microprogram version, Phase 1 is meant to be executed once, Phase 2 any number of times. Figure P2 indicates the logical flow of the system for a case in which both phases are executed. The figure also indicates the inputs and outputs of each program, including the data base files created and used by each program.

P4.2 Program Descriptions

The Timing Analyzer shall be comprised of the three computer programs identified in Section P4.1. Design data for each program follow.

P4.2.1 MULTI Source Code Analyzer Program

The MULTI Source Code Analyzer shall fulfill the requirements specified in Section P2.2.1. Additional design data follow.

P4.2.1.1 Inputs: The inputs to the program shall consist of the source code for a microprogram written in MULTI. This source code may be input from punched cards, magnetic tape, or disk. Each logical record in the file shall be an 80-character image of the format specified in the Nanodata Corporation report "MULTI Micromachine Descripton" (2 February 1976).

P4.2.1.2 Outputs: The output of the program shall consist of control flow tables, described in Section P4.2.1.3, and a numbered listing of the MULTI microprogram. The numbered listing shall be directed to a line

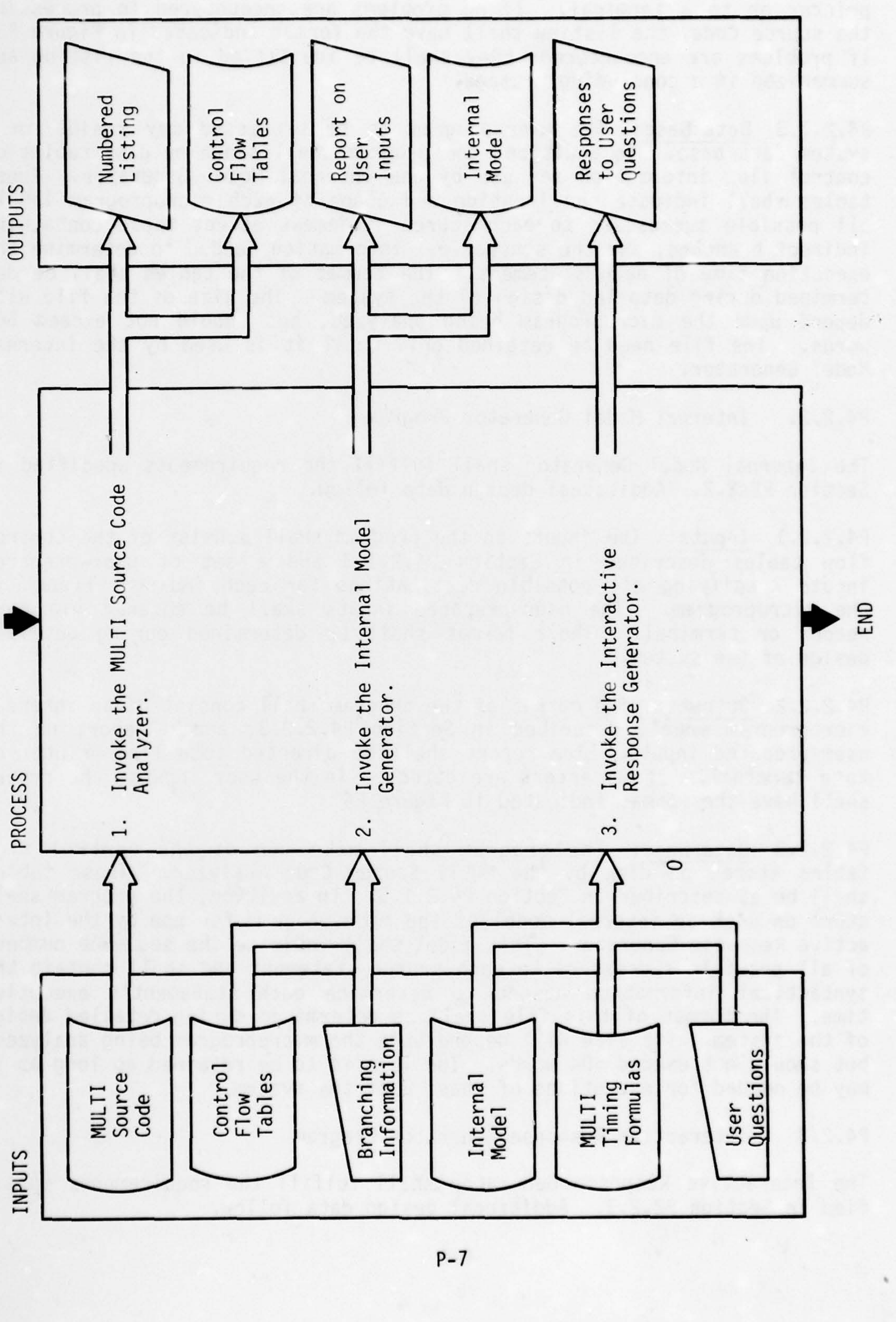


Figure P2. System Logical Flow for the Timing Analyzer

printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure P3. If problems are encountered, they shall be identified in the listing and summarized in a concluding message.

P4.2.1.3 Data Base: The microprogram to be processed may reside in a system data base. In addition, the program shall store on disk tables of control flow information for use by the Internal Model Generator. These tables shall indicate the location and usage of each microprogram label, all possible successors to each source statement except those containing indirect branches, and the syntactical information needed to determine the execution time of each statement. The format of the tables shall be determined during detailed design of the system. The size of the file will depend upon the microprogram being analyzed, but should not exceed 50K words. The file need be retained only until it is used by the Internal Model Generator.

P4.2.2. Internal Model Generator Program

The Internal Model Generator shall fulfill the requirements specified in Section P2.2.2. Additional design data follow.

P4.2.2.1 Inputs: The inputs to the program shall consist of the control flow tables described in Section P4.2.1.3 and a set of user-prepared inputs specifying all possible destinations for each indirect branch in the microprogram. The user-prepared inputs shall be entered via card reader or terminal. Their format shall be determined during detailed design of the system.

P4.2.2.2 Outputs: The output of the program shall consist of an internal microprogram model, described in Section P4.2.2.3, and a report on the user-prepared inputs. The report shall be directed to a line printer or to a terminal. If no errors are detected in the user inputs, the report shall have the format indicated in Figure P4.

P4.2.2.3 Data Base: The program shall make use of the control flow tables stored on disk by the MULTI Source Code Analyzer. These tables shall be as described in Section P4.2.1.3. In addition, the program shall store on disk an internal model of the microprogram for use by the Interactive Response Generator. This model shall indicate the sequence numbers of all possible successors to each source statement and shall contain the syntactical information needed to determine each statement's execution time. The format of this file shall be determined during detailed design of the system. Its size will depend upon the microprogram being analyzed, but should not exceed 50K words. The file is to be retained as long as it may be needed for executions of Phase 2 of the system.

P4.2.3 Interactive Response Generator Program

The Interactive Response Generator shall fulfill the requirements specified in Section P2.2.3. Additional design data follow.

```
STATEMENT #          SOURCE STATEMENT

   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     .                            .
     .                            .
     .                            .
```

Figure P3. Format of the Numbered Source Listing

```
STATMENT #          DESTINATIONS OF INDIRECT BRANCHES

XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX               XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                          .
  .                          .
  .                          .
```

Figure P4. Format of the Report on User Inputs

P4.2.3.1 <u>Inputs</u>: The inputs to the program shall consist of the internal microprogram model described in Section P4.2.2.3, a control file of timing formulas, described in Section P4.2.3.3, and a set of user questions concerning the execution time of selected microprogram paths. The information to be provided in each question shall be as specified in Section P2.2.3. The format of the questions shall be determined during detailed design of the system. They shall be input via card reader or terminal.

P4.2.3.2 <u>Outputs</u>: The output of the program shall consist of responses to user questions. Each response shall include a path description identifying the sequence of statements encountered on the user-specified path and a report of the desired execution time(s) for the path. The responses shall be directed to a line printer or to a terminal. Their format shall be as indicated in Figure P5.

P4.2.3.3 <u>Data Base</u>: The program shall make use of the internal microprogram model stored on disk by the Internal Model Generator. This model shall be as described in Section P4.2.2.3. The program shall also make use of a control file of timing formulas for MULTI microinstructions. This file shall be created as part of the Timing Analyzer system. It shall be stored on disk, requiring approximately 5,000 words of storage. Its format shall be determined during detailed design of the system. It is to be retained as long as the Timing Analyzer is being used.

```
MICROPROGRAM PATH

  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

MIN TIME   XXXX

MAX TIME   XXXX

REPRESENTATIVE TIME   XXXX

SPECIFIC TIME   XXXX
```

Figure P5. Format of the Responses to User Questions

APPENDIX Q
FUNCTIONAL DESCRIPTION FOR THE DATA FLOW ANALYZER

Q1. GENERAL

Q1.1 Purpose of Functional Description

See Appendix G.

Q1.2 Project References

See Appendix G.

Q2. SYSTEM SUMMARY

Q2.1 Background

See Appendix G.

Q2.2 Objectives

The purpose of the Data Flow Analyzer is to provide an automated means of investigating the flow of data within a microprogram written in SMITE or MULTI. The system will be capable of detecting data flow anomalies and of responding interactively to user questions regarding data relationships in the microprogram. It is intended to be used during microprogram debugging and maintenance activities.

Q2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of a Data Flow Analyzer. Analysis of a microprogram to detect data flow anomalies and to identify data relationships is a manual process.

Q2.4 Proposed Methods and Procedures

The proposed method for investigating the flow of data within a microprogram is to submit the microprogram to the Data Flow Analyzer. The system will operate in two phases, either of which may be executed independently. Phase 1 will analyze the microprogram source code, generate a numbered microprogram listing, report on user inputs, and build an internal model of the microprogram. Phase 2 will use the internal model to generate a report identifying data flow anomalies and to respond interactively to user questions concerning data relationships in the microprogram. Figure Q1 illustrates the data flow for the system. Figure Q2 shows its major processing steps.

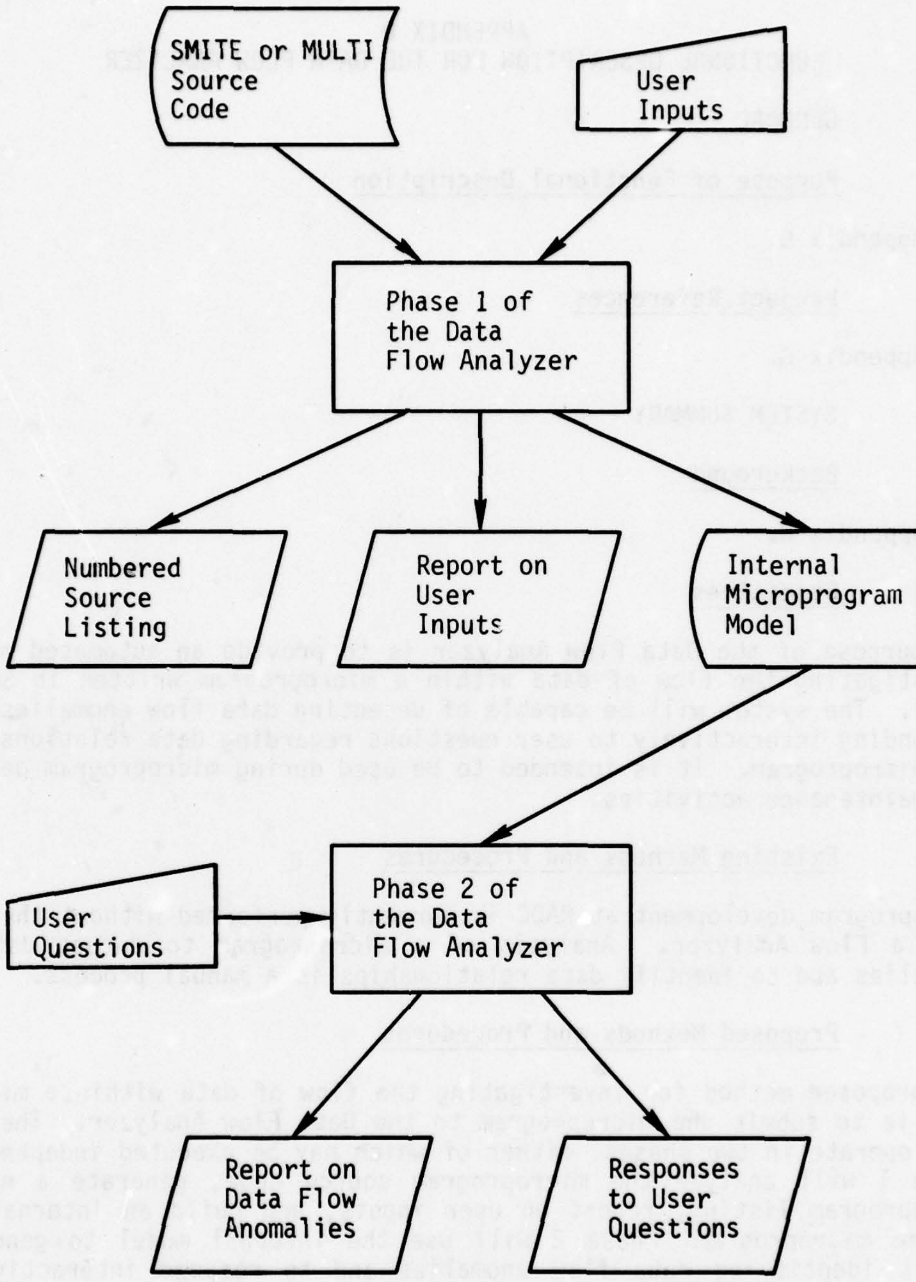

SMITE or MULTI Source Code
User Inputs
Phase 1 of the Data Flow Analyzer
Numbered Source Listing
Report on User Inputs
Internal Microprogram Model
User Questions
Phase 2 of the Data Flow Analyzer
Report on Data Flow Anomalies
Responses to User Questions

Figure Q1. Data Flow for the Data Flow Analyzer

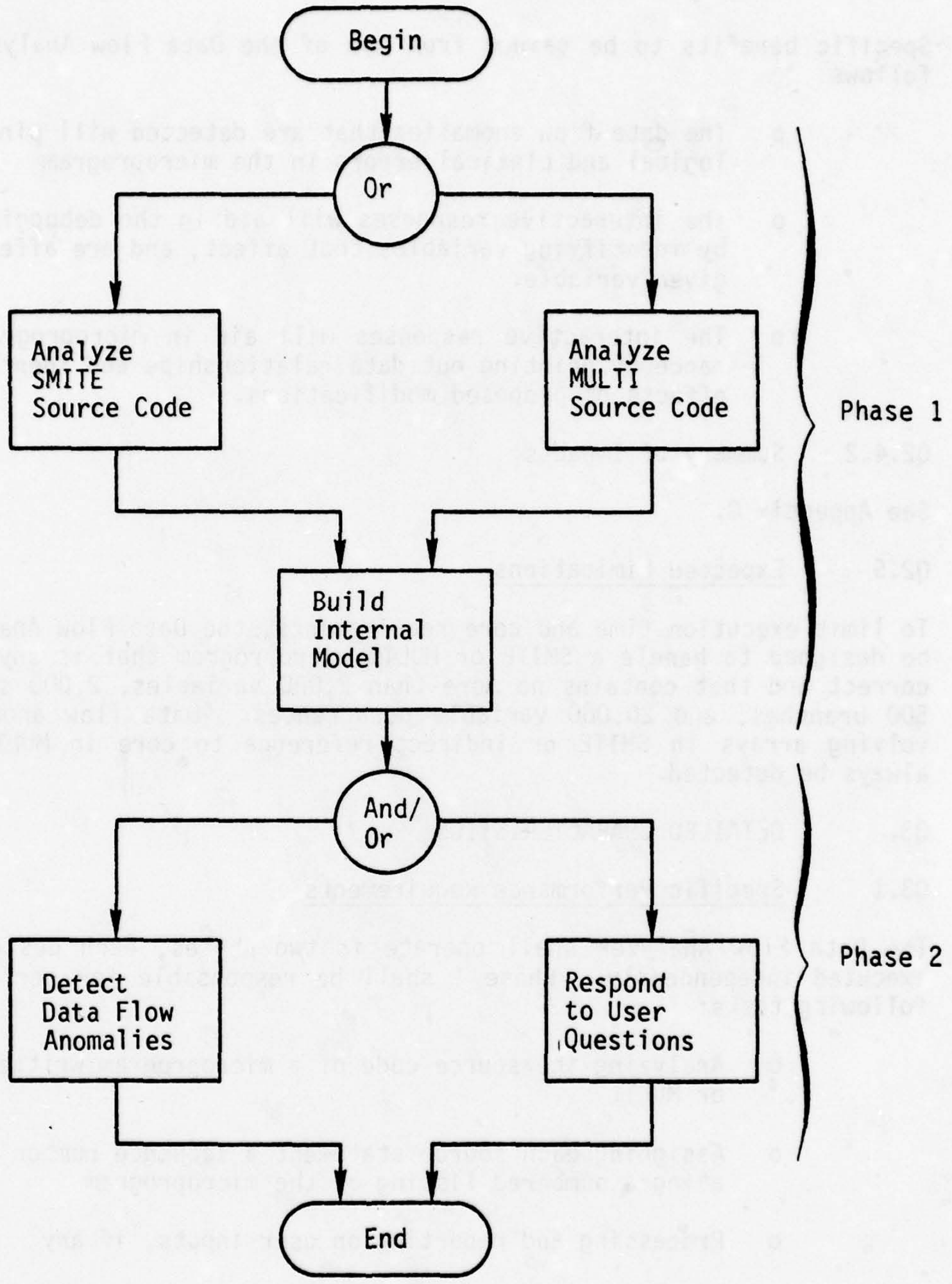

Begin
Or
Analyze SMITE Source Code
Analyze MULTI Source Code
Phase 1
Build Internal Model
And/ Or
Detect Data Flow Anomalies
Respond to User Questions
Phase 2
End

Figure Q2. Major Processing Steps in the Data Flow Analyzer

Q2.4.1 Summary of Improvements

Specific benefits to be gained from use of the Data Flow Analyzer are as follows:

- o The data flow anomalies that are detected will pinpoint both logical and clerical errors in the microprogram
- o The interactive responses will aid in the debugging process by identifying variables that affect, and are affected by, a given variable.
- o The interactive responses will aid in microprogram maintenance by pointing out data relationships and identifying the effects of proposed modifications.

Q2.4.2 Summary of Impacts

See Appendix G.

Q2.5 Expected Limitations

To limit execution time and core requirements, the Data Flow Analyzer will be designed to handle a SMITE or MULTI microprogram that is snytactically correct and that contains no more than 2,000 variables, 2,000 statements, 500 branches, and 20,000 variable occurrences. Data flow anomalies involving arrays in SMITE or indirect reference to core in MULTI may not always be detected.

Q3. DETAILED CHARACTERISTICS

Q3.1 Specific Performance Requirements

The Data Flow Analyzer shall operate in two phases, each designed to be executed independently. Phase 1 shall be responsible for performing the following tasks:

- o Analyzing the source code of a microprogram written in SMITE or MULTI
- o Assigning each source statement a sequence number and generating a numbered listing of the microprogram
- o Processing and reporting on user inputs, if any
- o Building an internal model of the microprogram

Phase 1 shall consist of two steps. In the first step, the source code shall be analyzed, the numbered listing shall be generated, and data flow information shall be stored in tables. This information shall indicate both the flow of control within the microprogram and the data usage properties of each microprogram statement. If any source statement cannot be interpreted during this step, it shall be identified in the numbered listing, and the system shall proceed to the next statement. If any of the numerical limitations given in Section Q2.5 is exceeded during the processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

The second step of Phase 1 shall consist of building an internal model of the microprogram, using the information stored in tables during Step 1. For a SMITE program, this step shall proceed directly from the first without user intervention. For a MULTI microprogram, this step will require user-supplied inputs specifying all possible destinations for indirect branches. These user inputs shall make use of the sequence numbers provided in the numbered listing produced in Step 1. The system shall process the user inputs, check them for errors in format and content, and prepare a report on them, identifying any errors detected. If errors are present, the system shall request and process corrections. When the inputs are correct, the system shall use them to aid in building the internal model. This model shall indicate all possible predecessors and successors to each source statement and shall specify the data usage properties of each statement. The internal model shall be stored on disk to be used as input to Phase 2.

In Phase 2, the system shall use the internal model of the microprogram to generate data flow information for the user. Two types of output shall be provided:

- o A report on data flow anomalies
- o Interactive responses to user questions about data relationships in the microprogram

Each of these outputs shall be optional. The types of data flow anomalies that shall be detected are as follows:

- o A variable has been used before it is set
- o A variable has been set but never used
- o A variable has been reset before its previous value was used

In searching for these anomalies, the system shall process the microprogram through routine boundaries and shall correctly handle variable overlays, equivalenced variables, and parameter correspondence. The data flow anomalies shall be detected in all cases except those in which array references in SMITE or indirect references to core in MULTI preclude static detection.

In addition to detecting data flow anomalies, the system shall be capable of responding interactively to user-supplied questions of the following types:

- o What input variables directly affect a given variable?
- o What non-input variables directly affect a given variable?
- o What input variables indirectly affect a given variable?
- o What non-input variables indirectly affect a given variable?
- o What output variables are directly affected by a given variable?
- o What non-output variables are directly affected by a given variable?
- o What output variables are indirectly affected by a given variable?
- o What non-output variables are indirectly affected by a given variable?

A variable shall be said to have a direct effect on a second variable if its value is used in computations that determine the value of the second variable; it shall be said to have an indirect effect if its value is used in branching decisions that affect action taken on the second variable.

Q3.1.1 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

Q3.1.2 Timing

There are no timing requirements on the system.

Q3.2 System Functions

The Data Flow Analyzer shall perform five major functions, each of which shall be implemented as a separate program in the system. The functions shall be as follows:

- o SMITE Source Code Analysis
- o MULTI Source Code Analysis
- o Internal Model Generation
- o Anomaly Detection
- o Interactive Response Generation

The SMITE and MULTI Source Code Analysis functions shall each fulfill the requirements described in Section Q3.1 for Phase 1, Step 1 of the system. The Internal Model Generation function shall fulfill the requirements of Phase 1, Step 2. The requirements specified for Phase 2 of the system shall be fulfilled by the Anomaly Detection and Interactive Response Generation functions.

Q3.3 Inputs/Outputs

Q3.3.1 Inputs

The inputs to the Data Flow Analyzer shall consist of the following:

- o The source code for a microprogram written in SMITE or MULTI
- o User inputs identifying all possible destinations for indirect branches in a MULTI microprogram
- o User questions concerning data relationships in the microprogram

The source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. The two types of user inputs may be entered via card reader or terminal. The user inputs identifying destinations of indirect branches shall make use of the sequence numbers provided in the numbered listing generated by the system. The user questions shall be of the types specified in Section Q3.1. The format of the user inputs will be determined during detailed design of the system.

Q3.3.2 Outputs

The output of the system shall consist of the following:

- o A numbered listing of the microprogram
- o A report on the Phase 1 user inputs specifying destinations of indirect branches
- o A report on data flow anomalies, if requested
- o Responses to user questions about data relationships in the microprogram

The formats of these outputs shall be as indicated in Figures Q3, Q4, Q5, and Q6, respectively. Each output shall be directed to a file that may be output on a line printer or on a terminal.

Q3.4 Data Characteristics

The microprogram to be processed may reside in a system data base. In addition, the system shall create the following files to be stored on disk:

- o The tables of data flow information
- o The internal model for the microprogram

The size of these files will depend upon the microprogram being analyzed, but the maximum storage required for each file will be approximately 50K

```
STATEMENT #          SOURCE STATEMENT

   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     .                   .
     .                   .
     .                   .
```

Figure Q3. Format of the Numbered Source Listing

```
STATMENT #        DESTINATIONS OF INDIRECT BRANCHES

XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                        .
  .                        .
  .                        .
```

Figure Q4. Format of the Report on User Inputs

```
                    DATA FLOW ANOMALIES

VARIABLE    ANOMALY DESCRIPTION

XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                                     .
   .                                     .
   .                                     .
```

Figure Q5. Format of the Report on Data Flow Anomalies

```
SELECTED    VARIABLES
VARIABLE    DIRECTLY AFFECTED

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

SELECTED    VARIABLES
VARIABLE    INDIRECTLY AFFECTED

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

SELECTED    VARIABLES DIRECTLY
VARIABLE    AFFECTING SELECTED VARIABLE

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

SELECTED    VARIABLES INDIRECTLY
VARIABLE    AFFECTING SELECTED VARIABLE

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

XXXXXXXX    XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
            XXXXXXXX  XXXXXXXX      .         .         .

    .           .

    .           .

    .           .
```

Figure Q6. Format of the Responses to User Questions

words. The first file, produced by the SMITE or MULTI Source Code Analysis function, need be saved only until it is used to generate the internal model for the microprogram. The second file is to be retained as long as it may be needed for executions of Phase 2 of the system.

Q3.5 Failure Contingencies

Not applicable.

Q4. ENVIRONMENT

Q4.1 Equipment Environment

The Data Flow Analyzer will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 120K words of main memory. The system will be stored on disk. The microprogram to be analyzed may be stored on punched cards, magnetic tape, or disk. The user inputs may be entered via card reader or terminal. The tables of data flow information and the internal microprogram model will be stored on disk. The outputs of the system will be directed to a line printer or to a terminal. No new equipment will be required.

Q4.2 Support Software Environment

Se Appendix G.

Q4.3 Interfaces

The system will not interface with any other system.

Q4.4 Security

The system will have no classified components.

Q5. COST FACTORS

Development of the Data Flow Analyzer will require approximately 35 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the system.

6. DEVELOPMENTAL PLAN

The Data Flow Analyzer is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Data Flow Analyzer is given in Figure Q7.

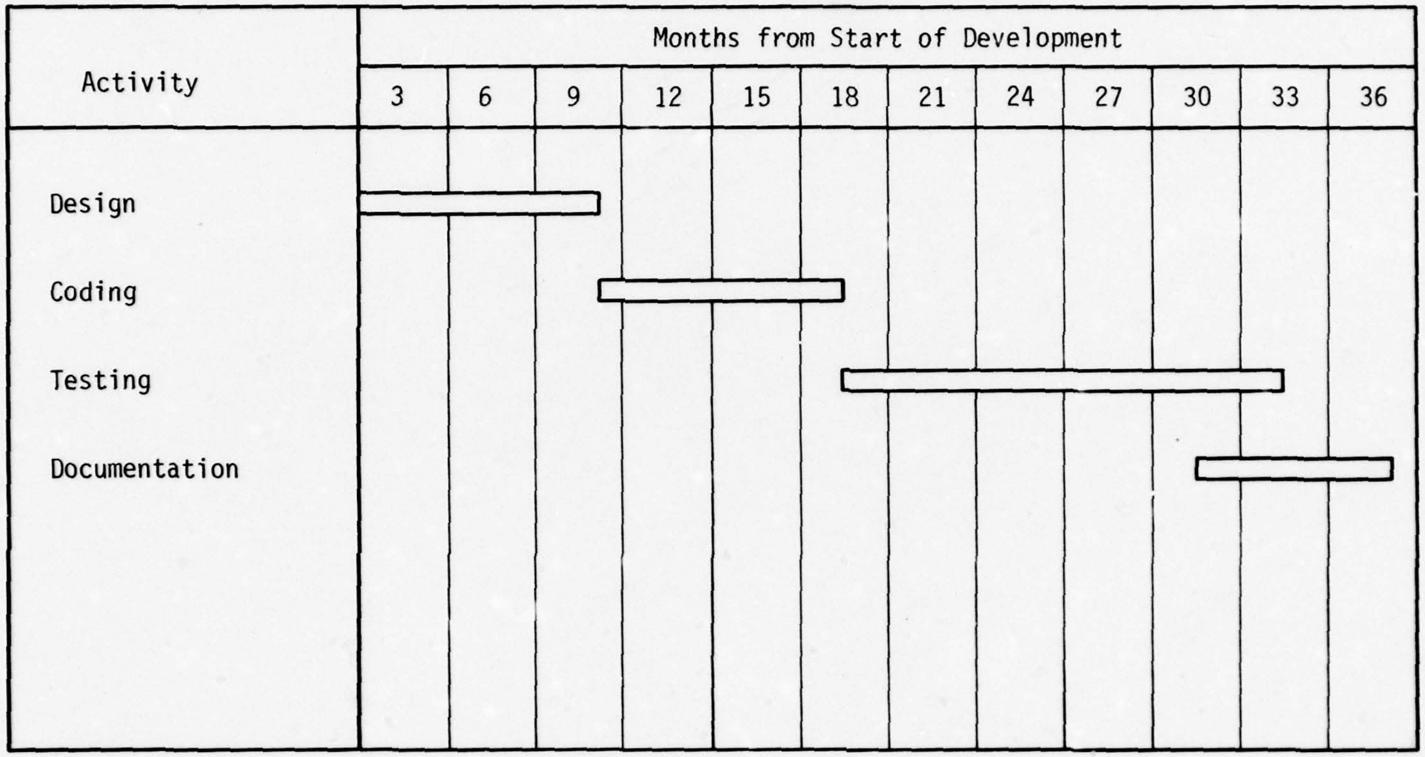


Figure Q7. Development Schedule for the Data Flow Analyzer

APPENDIX R
SYSTEM SPECIFICATION FOR THE DATA FLOW ANALYZER

R1. GENERAL

R1.1 Purpose of the System Specification

See Appendix H.

R1.2 Project References

See Appendix H.

R2. SUMMARY OF REQUIREMENTS

R2.1 System Description

The Data Flow Analyzer shall provide an automated means of investigating the flow of data within a microprogram written in SMITE or MULTI. The system shall operate in two independent phases. Phase 1 shall analyze the microprogram source code, generate a numbered microprogram listing, report on user inputs, and build an internal model of the microprogram. Phase 2 shall use the internal model to generate a report identifying data flow anomalies and to respond interactively to user questions concerning data relationships in the microprogram.

The system shall consist of five computer programs. Figure R1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

R2.2 System Functions

The Data Flow Analyzer shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o SMITE Source Code Analysis
- o MULTI Source Code Analysis
- o Internal Model Generation
- o Anomaly Detection
- o Interactive Response Generation

The following paragraphs identify the requirements to be satisfied by each of these functions.

R2.2.1 SMITE Source Code Analysis Function

The SMITE Source Code Analysis function shall accept as input the source code of a program written in SMITE. The function shall process each source statement of the program as follows:

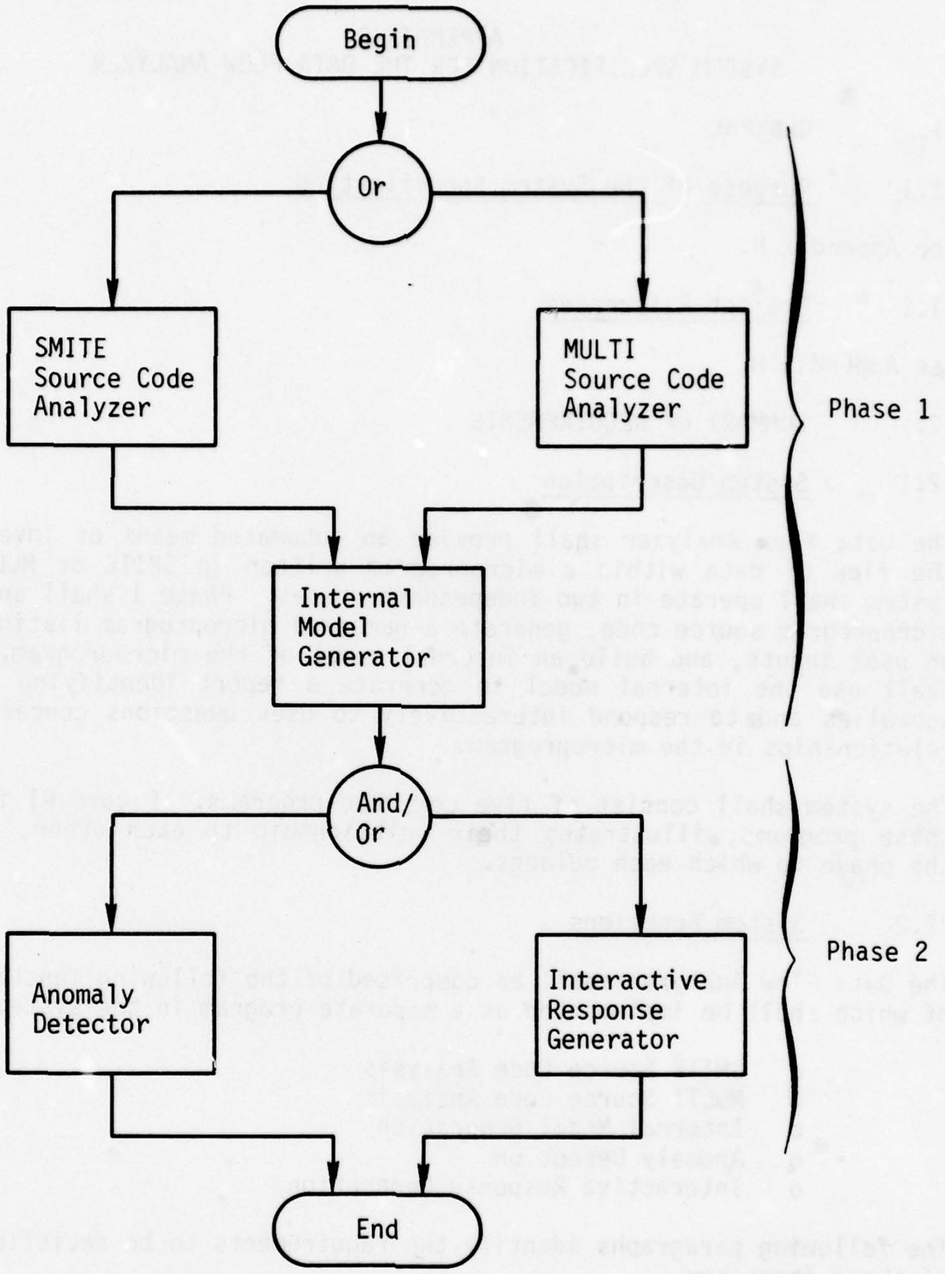

Begin
Or
SMITE Source Code Analyzer
MULTI Source Code Analyzer
Phase 1
Internal Model Generator
And/ Or
Anomaly Detector
Interactive Response Generator
Phase 2
End

Figure R2. Relationship of the Components of the Data Flow Analyzer

- Assign the statement a sequence number
- Output a copy of the statement and its sequence number in a listing of the program
- Analyze the contents of the statement and store relevant information in data flow tables

Source code comments shall be copied into the program listing without being analyzed or assigned a sequence number. Information to be stored in the data flow tables shall include the following:

- The location and usage of each program label
- All possible predecessors and successors to each source statement
- The data usage properties of each statement

To limit execution time and core requirements, the function shall expect the program to be syntactically correct and to have no more than 2,000 variables, 2,000 statements, 500 branches, and 20,000 variable occurrences. If any source statement cannot be interpreted, the function shall report this problem in the program listing and shall proceed to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, then continue to analyze and list each source statement, storing all relevant information for which space remains.

At the conclusion of the source code analysis, the function shall output a problem summary message if any source statements could not be interpreted or if space limitations were exceeded during the processing. The data flow tables shall be stored on disk for use by the Internal Model Generation function.

R2.2.2 MULTI Source Code Analysis Function

The requirements for the MULTI Source Code Analysis function shall be the same as those given in Section R2.2.1 for the SMITE Source Code Analysis function, with two exceptions:

- The function shall accept as input the source code of a microprogram written in MULTI.
- Possible successors to statements containing indirect branches, which cannot be determined from the source code, shall be left undetermined in the data flow tables.

R2.2.3 Internal Model Generation Function

The Internal Model Generation function shall accept as input the data flow tables produced by the SMITE or MULTI Source Code Analysis function and, in the case of a MULTI microprogram, a set of user-prepared inputs specifying all possible destinations for the indirect branches in the microprogram. If user inputs are present, the function shall check them for errors in format and content, record the specified branch destinations, and prepare a report that lists each user input and identifies any errors detected. If errors have been detected or if the inputs do not provide all of the required information, the function shall request and process the required corrections and additions. When the inputs are correct and complete, the function shall generate an internal model of the microprogram. This model shall indicate the sequence numbers of all possible predecessors and successors of each source statement and shall specify the data usage properties of each statement. The model shall be stored on disk for use by the Anomaly Detection and Interactive Response Generation functions.

R2.2.4 Anomaly Detection Function

The Anomaly Detection function shall accept as input the internal model produced by the Internal Model Generation function. It shall analyze this model to identify anomalies of the following types:

- o A variable has been used before it is set
- o A variable has been set but never used
- o A variable has been reset before its previous value was used

In detecting these anomalies, the function shall be capable of processing the microprogram through routine boundaries and shall correctly handle variable overlays, equivalenced variables, and parameter correspondence. The anomalies identified above shall be detected in all cases except those in which array references in SMITE or indirect references to core in MULTI preclude detection during static analysis. The function shall prepare a report identifying all of the anomalies that have been detected.

R2.2.5 Interactive Response Generation Function

The Interactive Response Generation function shall accept as input the internal model produced by the Internal Model Generation function and a set of user questions concerning data relationships in the microprogram. The following types of questions shall be handled:

- o What input variables directly affect a given variable?
- o What non-input variables directly affect a given variable?
- o What input variables indirectly affect a given variable?

- o What non-input variables indirectly affect a given variable?
- o What output variables are directly affected by a given variable?
- o What non-output variables are directly affected by a given variable?
- o What output variables are indirectly affected by a given variable?
- o What non-output varibles are indirectly affected by a given variable?

A variable shall be said to have a direct effect on a second variable if its value is used in computations that determine the value of the second variable; it shall be said to have an indirect effect if its value is used in branching decisions that affect action taken on the second variable.

The function shall accept each question, permit the user to make corrections if necessary, analyze the internal model to determine the appropriate response, and output this response.

R2.3 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

R2.4 Timing

There are no timing requirements on the system.

R2.5 Flexibility

The system shall be designed in such a way that the limit on the number of variables, statements, branches, and variable occurrences can be easily modified.

R3. ENVIRONMENT

R3.1 Equipment Environment

See Appendix H for equipment description.

The Data Flow Analyzer will require approximately 120K words of main memory. The system will be stored on disk, requiring approximately 60K words of storage for the source code, object code, and load module. The user-prepared inputs may be entered via card reader or terminal. The microprogram source code may be input from punched cards, magnetic tape, or disk. The system-generated files shall be stored on disk, requiring a maximum of 50K words of storage each. The outputs may be directed to a line printer or to a terminal. No new equipment will be required.

R3.2 Support Software Environment

See Appendix H.

R3.3 Interfaces

The system shall not interface with any other system.

R3.4 Security

The system shall have no classified components.

R3.5 Controls

No controls shall be established by the system.

R4. DESIGN DATA

R4.1 System Logical Flow

The Data Flow Analyzer shall be a system consisting of five computer programs:

- o The SMITE Source Code Analyzer
- o The MULTI Source Code Analyzer
- o The Internal Model Generator
- o The Anomaly Detector
- o The Interactive Response Generator

The first three of these programs shall constitute Phase 1 of the system. To execute this phase, the user would invoke either the SMITE or MULTI Source Code Analyzer, then invoke the Internal Model Generator. The other two programs shall comprise Phase 2 of the system. To execute this phase, the user would invoke one or both programs. If both are invoked, their order of execution is immaterial. For a given microprogram version, Phase 1 is meant to be executed once, Phase 2 any number of times. Figure R2 indicates the logical flow of the system for a case in which both phases are executed and the Anomaly Detector is invoked before the Interactive Response Generator. The figure also indicates the inputs and outputs of each program, including the data base files created and used by each program.

R4.2 Program Descriptions

The Data Flow Analyzer shall be comprised of the five computer programs identified in Section R4.1. Design data for each program follow.

R4.2.1 SMITE Source Code Analyzer Program

The SMITE Source Code Analyzer shall fulfill the requirements specified in Section R2.2.1. Additional design data follow.

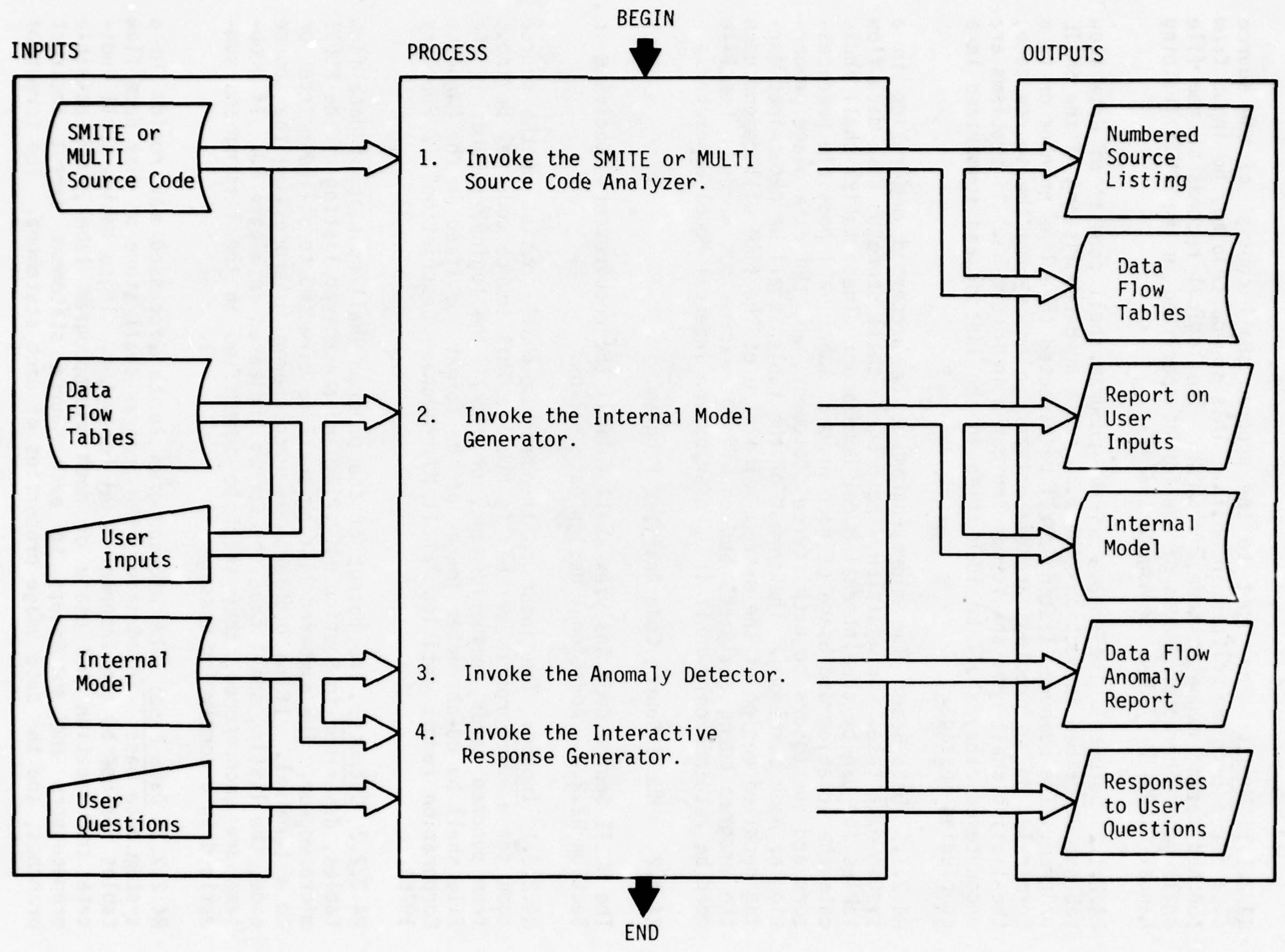

BEGIN
INPUTS
PROCESS
OUTPUTS
SMITE or MULTI Source Code
Data Flow Tables
User Inputs
Internal Model
User Questions
1. Invoke the SMITE or MULTI Source Code Analyzer.
2. Invoke the Internal Model Generator.
3. Invoke the Anomaly Detector.
4. Invoke the Interactive Response Generator.
Numbered Source Listing
Data Flow Tables
Report on User Inputs
Internal Model
Data Flow Anomaly Report
Responses to User Questions
END

Figure R2. Typical Logical Flow for the Data Flow Analyzer

R4.2.1.1 Inputs: The input to the program shall consist of the source code for a program written in SMITE. This source code may be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

R4.2.1.2 Outputs: The output of the program shall consist of data flow tables, described in Section R4.2.1.3, and a numbered listing of the SMITE program. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure R3. If problems are encountered, they shall be identified in the listing and summarized in a concluding message.

R4.2.1.3 Data Base: The microprogram to be processed may reside in a system data base. In addition, the program shall store on disk data flow tables for use by the Internal Model Generator. These tables shall indicate the location and usage of each program label, all possible predecessors and successors to each source statement, and the data usage properties of each statement. The format of the tables shall be determined during detailed design of the system. The size of the file will depend upon the program being analyzed, but should not exceed 50K words. The file need be retained only until it is used by the Internal Model Generator.

R4.2.2 MULTI Source Code Analyzer Program

The MULTI Source Code Analyzer shall fulfill the requirements specified in Section R2.2.2. Additional design data follow.

R4.2.2.1 Inputs: The input to the program shall consist of the source code for a microprogram written in MULTI. This source code may be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the Nanodata Corporation report entitled "MULTI Micromachine Description" (2 February 1976).

R4.2.2.2 Outputs: The output of the program shall consist of data flow tables, described in Section R4.2.2.3, and a numbered listing of the MULTI microprogram. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure R3. If problems are encountered, they shall be identified in the listing and summarized in a concluding message.

R4.2.2.3 Data Base: The microprogram to be processed may reside in a system data base. In addition, the program shall store on disk data flow tables for use by the Internal Model Generator. These tables shall indicate the location and usage of each microprogram label, all possible predecessors and successors to each source statement except indirect branches, and the data usage properties of each statement. The format of

```
STATEMENT #             SOURCE STATEMENT

 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                      .
   .                      .
   .                      .
```

Figure R3. Format of the Numbered Source Listing

the tables shall be determined during detailed design of the system. The size of the file will depend upon the microprogram being analyzed but should not exceed 50K words. The file need be retained only until it is used by the Internal Model Generator.

R4.2.3 Internal Model Generator Program

The Internal Model Generator shall fulfill the requirements specified in Section R2.2.3. Additional design data follow.

R4.2.3.1 Inputs: For a SMITE program, the inputs to the program shall consist of data flow tables described in Section R4.2.1.3. For a MULTI microprogram, the inputs shall consist of the data flow tables described in Section R4.2.2.3 plus a set of user-prepared inputs specifying all possible destinations for each indirect branch in the microprogram. The user-prepared inputs shall be entered via card reader or terminal. Their format shall be determined during detailed design of the system.

R4.2.3.2 Outputs: The output of the program shall consist of an internal microprogram model, described in Section R4.2.3.3, and a report on the user-prepared inputs. The report shall be directed to a line printer or to a terminal. If the user inputs are correct and complete, the report shall have the format indicated in Figure R4.

R4.2.3.3 Data Base: The program shall make use of the tables of data flow information stored on disk by the SMITE or MULTI Source Code Analyzer. These tables shall be as described in Sections R4.2.1.3 and R4.2.2.3, respectively. The program shall store on disk an internal model of the microprogram for use by the Anomaly Detector and the Interactive Response Generator. This model shall indicate the sequence numbers of all possible predecessors and successors to each source statement and shall specify the data usage properties of each statement. The format of the model shall be determined during detailed design of the system. Its size will depend upon the microprogram being analyzed, but should not exceed 50K words. The file is to be retained as long as it may be needed for executions of Phase 2 of the system.

R4.2.4 Anomaly Detector Program

The Anomaly Detector shall fulfill the requirements specified in Section R2.2.4. Additional design data follow.

R4.2.4.1 Inputs: The input to the program shall consist of the internal microprogram model described in Section R4.2.3.3.

R4.2.4.2 Outputs: The output of the program shall consist of a report identifying data flow anomalies in the microprogram. This report shall be directed to a line printer or to a terminal. Its format shall be as indicated in Figure R5.

```
STATMENT #        DESTINATIONS OF INDIRECT BRANCHES

XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
XXXXX             XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX  XXXXX
  .                         .
  .                         .
  .                         .
```

Figure R4. Format of the Report on User Inputs

```
                  DATA FLOW ANOMALIES

VARIABLE    ANOMALY DESCRIPTION

XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    .                                   .
    .                                   .
    .                                   .
```

Figure R5. Format of the Report on Data Flow Anomalies

R4.2.4.3 Data Base: The program shall make use of the internal microprogram model stored on disk by the Internal Model Generator. This model shall be as described in Section R4.2.3.3.

R4.2.5 Interactive Response Generator Program

The Interactive Response Generator shall fulfill the requirements specified in Section R2.2.5. Additional design data follow.

R4.2.5.1 Inputs: The inputs to the program shall consist of the internal model described in Section R4.2.3.3 and a set of user-prepared questions concerning data relationships in the microprogram. The types of questions that may be input shall be as specified in Section R2.2.5. Their format shall be determined during detailed design of the system. They shall be input via card reader or terminal.

R4.2.5.2 Outputs: The output of the program shall consist of responses to user questions. These responses shall be directed to a line printer or to a terminal. Their format shall be as indicated in Figure R6. If a user question cannot be interpreted, an error message shall be output in place of a response, enabling the user to correct and resubmit the question.

R4.2.5.3 Data Base: The program shall make use of the internal model stored on disk by the Internal Model Generator. This model shall be as described in Section R4.2.3.3.

```
SELECTED   VARIABLES
VARIABLE   DIRECTLY AFFECTED

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX .   .         .         .

SELECTED   VARIABLES
VARIABLE   INDIRECTLY AFFECTED

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

SELECTED   VARIABLES DIRECTLY
VARIABLE   AFFECTING SELECTED VARIABLE

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

SELECTED   VARIABLES INDIRECTLY
VARIABLE   AFFECTING SELECTED VARIABLE

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

XXXXXXXX   XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX  XXXXXXXX
           XXXXXXXX  XXXXXXXX     .         .         .

   .          .


   .          .


   .          .
```

Figure R6. Format of the Responses to User Questions

APPENDIX S
FUNCTIONAL DESCRIPTION FOR THE EXECUTION MONITOR

S1. GENERAL

S1.1 Purpose of Functional Description

See Appendix G.

S1.2 Project References

See Appendix G.

S2. SYSTEM SUMMARY

S2.1 Background

See Appendix G.

S2.2 Objectives

The purpose of the Execution Monitor is to provide an automated means of determining the execution characteristics of a program written in SMITE. The system will permit monitoring of a SMITE program by adding statistics-collecting instructions that will record the execution history of each code segment during execution of the program. The results will be reported to the user and maintained in a history file for comparative analysis.

S2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of an Execution Monitor. There is no automated means of collecting execution statistics for SMITE programs or of storing such statistics for comparative analysis.

S2.4 Proposed Methods and Procedures

The proposed method for determining the execution characteristics of a SMITE program is to submit the program to the Execution Monitor. The system will operate in two independent phases. Phase 1 will process and report on user inputs, analyze the source code and divide it into logical code segments, generate a file containing a listing of the segmented code, and instrument the source code with special instructions and statistics-collecting routines from a support file to be developed in conjunction with the system. At the conclusion of Phase 1, the instrumented source code may be executed, generating statistics that indicate the execution history of each code segment. Phase 2 of the system may then be invoked to process and report on user inputs, record the execution statistics in a history file, and prepare a series of reports on these and previously recorded statistics. Figure S1 illustrates the data flow for the system. Figure S2 shows its major processing steps.

S2.4.1 Summary of Improvements

The Execution Monitor will provide insight into the execution characteristics of a SMITE program. Specific benefits to be gained from the system are as follows:

- o The execution statistics will aid in program debugging by identifying the branches taken during program execution.
- o The system will aid in the generation of test cases by identifying execution patterns and untested segments of code.
- o The system will provide documentation that will permit analysis and evaluation of both individual and cumulative test runs.
- o The system will identify highly used code segments, which may be candidates for optimization, and unused code segments, which may be unreachable.

S2.4.2 Summary of Impacts

See Appendix G.

S2.5 Expected Limitations

To limit execution time and core requirements, the Execution Monitor will be designed to handle a SMITE program that is syntactically correct and that contains no more than 2,000 branches. It will store up to 999 sets of execution statistics in the history file, and up to 10 of these sets of statistics may be included in one report. Since the instrumented source code will contain more statements than the original, both the core requirements and execution time of the program will be increased to a degree dependent upon the number of branches instrumented. Some programs may increase to a point at which execution of the instrumented code is prohibited by the execution environment.

S3. DETAILED CHARACTERISTICS

S3.1 Specific Performance Requirements

The Execution Monitor shall operate in two phases. Phase 1 shall accept as input a set of user-prepared inputs and the source code of a program written in SMITE. The user-prepared inputs shall specify the following:

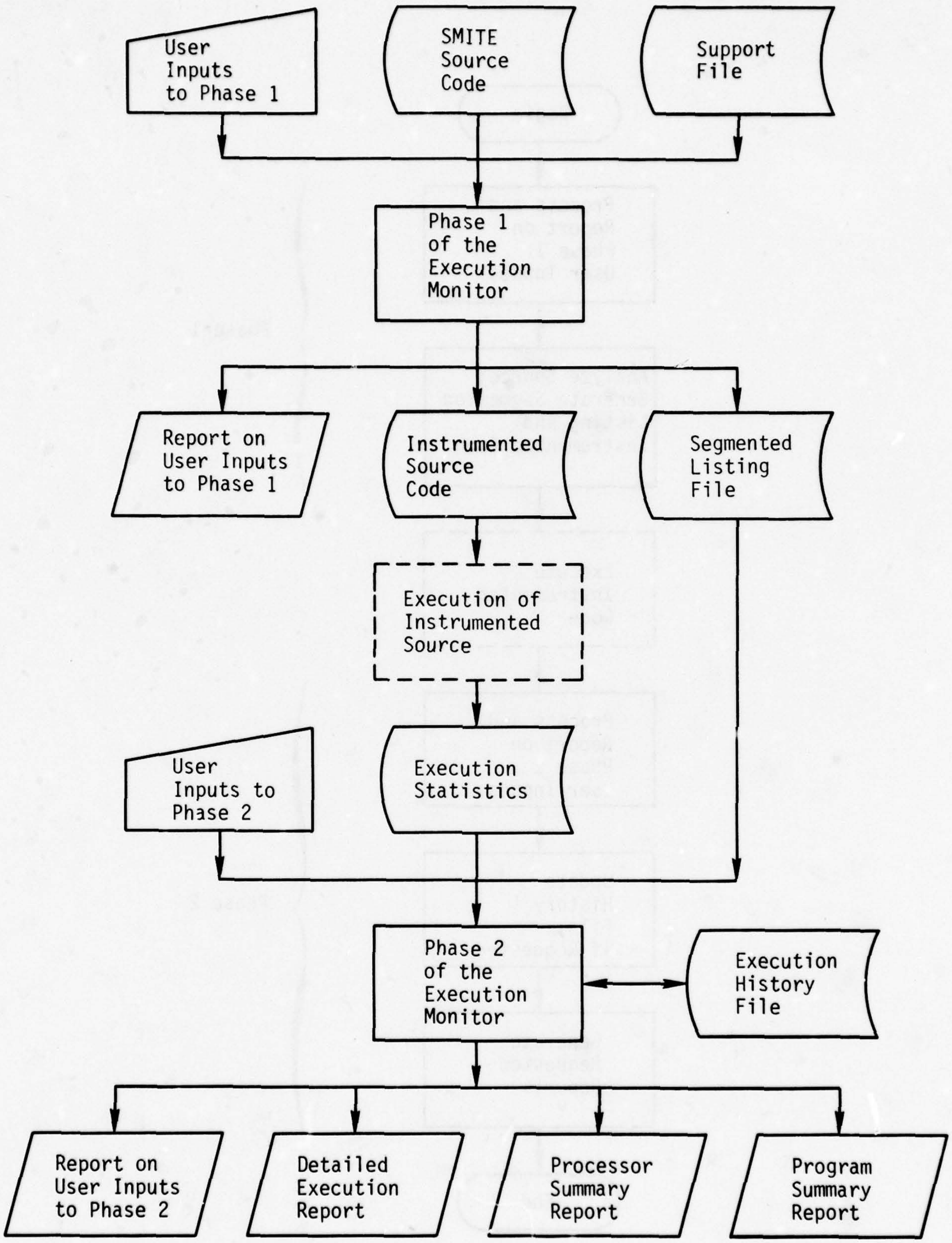


Figure S1. Data Flow for the Execution Monitor

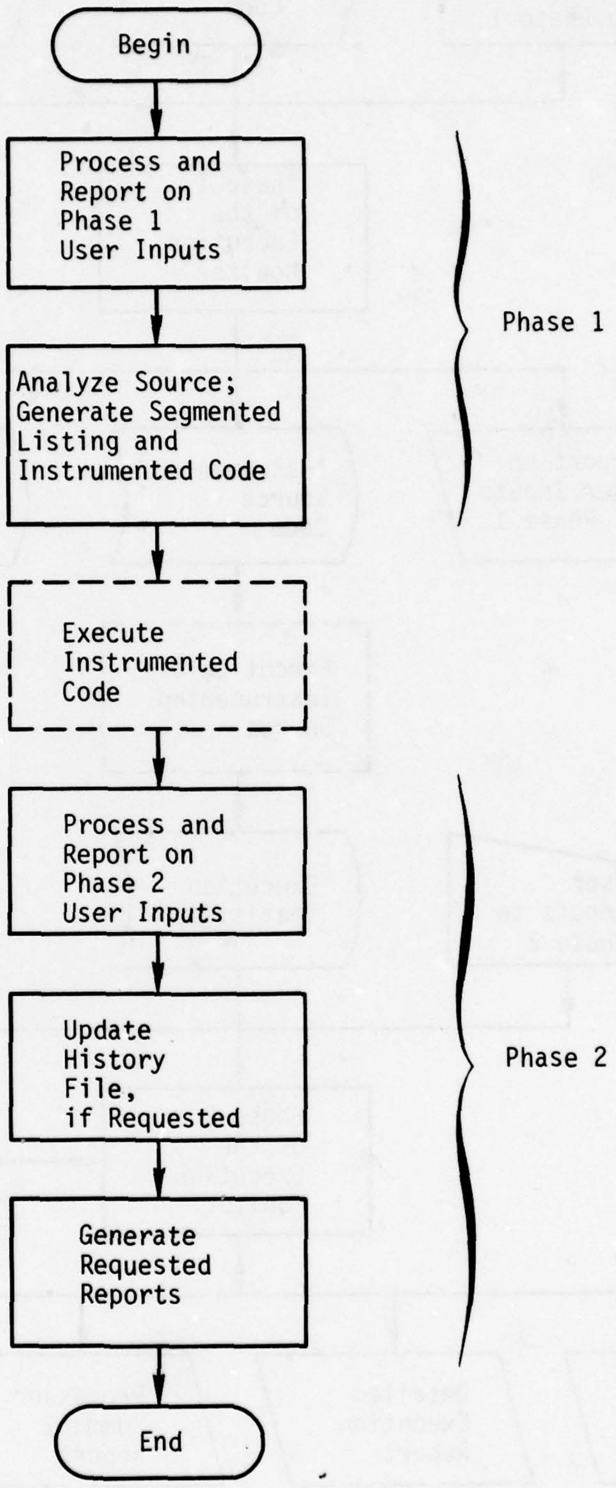

Begin
Process and Report on Phase 1 User Inputs
Analyze Source; Generate Segmented Listing and Instrumented Code
Execute Instrumented Code
Process and Report on Phase 2 User Inputs
Update History File, if Requested
Generate Requested Reports
End
Phase 1
Phase 2

Figure S2. Major Processing Steps of the Execution Monitor

- The processors that are to be monitored
- Whether the monitoring is to count the number of times each code segment is executed or simply identify those segments that have been executed at least once

Default values shall be provided for each specification. The system shall interpret the inputs, check them for errors in format and content, and prepare a report listing the inputs and identifying any errors detected in them. If errors are detected, the system shall request and process corrections. When the inputs are correct, the system shall proceed to process the SMITE source code.

Processing of the source code shall entail the following tasks:

- Dividing the selected processors into logical code segments
- Numbering the code segments and generating a segmented listing file containing a listing of the source code with each segment appropriately numbered
- Generating an instrumented version of the source code, i.e., a version that has been modified by addition of the following:
  - Special statements in each code segment to permit the execution history of that segment to be monitored
  - Appropriate declarations and initializing code for all monitoring variables
  - Statistics-collecting routines taken from a support file to be developed in conjunction with the system
  - Special instructions to cause the execution statistics to be written into a file

If any source statement cannot be interpreted during this processing, the function shall indicate this problem in the report on user inputs and shall proceed to the next source statement. If more than 2,000 code segments are encountered, the function shall indicate this problem in the report on user inputs, instrument 2,000 of the segments, and leave the remainder uninstrumented.

The modifications made to the source code shall be sufficient to cause a file of execution statistics to be generated during each execution of the instrumented source code. These statistics shall identify all code segments that have been executed in the instrumented processors or shall indicate the number of times each segment has been executed, depending upon user input to Phase 1.

Phase 2 shall accept the following as inputs:

- o A set of user-prepared inputs
- o The segmented listing file for an instrumented SMITE program
- o The file of execution statistics generated during one execution of the program
- o A history file of execution statistics

The user-prepared inputs shall specify the following:

- o The manner in which the history file is to be updated, if at all
- o The types of reports that are to be generated
- o The names of up to ten program executions whose statistics are to be used in the updating and reporting tasks

Default values shall be provided for each specification. The system shall interpret the inputs, check them for errors in format and content, store the specified options, and prepare a report listing the inputs and identifying any errors detected in them. If errors are detected, the system shall request and process corrections.

The system shall be capable of updating the history file in either of the following ways, depending upon user specification:

- o Deleting an entry from the file
- o Adding an entry containing the combined statistics from the selected program executions and assigning the entry a user-specified name

The system shall also be capable of generating the following types of reports, depending upon user specifications:

- o A Detailed Execution Report
- o A Processor Summary Report
- o A Program Summary Report

The Detailed Execution Report shall present the statistics from the execution statistics file. It shall contain a source listing of the monitored processors, identifying each code segment and indicating the execution history of that segment (i.e., the number of times the segment was executed or the fact that it was executed at least once). The Processor Summary Report shall indicate, for each monitored processor, the execution history of each code segment during the selected program executions and the percentage of statements and segments executed. The Program Summary Report shall indicate the percentage of all statements and segments executed in each monitored processor during the selected executions and the overall percentages for the monitored portion of the program.

S3.1.1 Accuracy and Validity

The percentages presented in the Processor Summary and Program Summary Reports shall be accurate to the nearest tenth of a percent.

S3.1.2 Timing

There are no timing requirements on the system.

S3.2 System Functions

The Execution Monitor shall perform two major functions, each of which shall be implemented as a separate program in the system. These functions shall be as follows:

- o Source Code Instrumentation
- o Execution Statistics Handling

The Source Code Instrumentation function shall fulfill the requirements described in Section S3.1 for Phase 1 of the system. The Execution Statistics Handling function shall fulfill the requirements specified for Phase 2.

S3.3 Inputs/Outputs

S3.3.1 Inputs

The inputs to the Execution Monitor shall consist of the following:

- o The source code for a program written in SMITE
- o User inputs to Phase 1
- o User inputs to Phase 2

The SMITE source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. The two sets of user inputs may be entered via card reader or terminal. The Phase 1 user inputs shall be as follows:

- o A list of the processors that are to be monitored
- o Specification of whether the monitoring is to count the number of times each code segment has been executed or is simply to identify those segments that have been executed at least once

The Phase 2 user inputs shall specify the following:

- o The manner in which the history file is to be updated, if at all

- o The types of reports that are to be generated
- o The names of up to 10 program executions whose statistics are to be used in the updating and/or reporting tasks (the execution covered by the execution statistics file shall be one of those selected)

The history file update specification may be either of the following:

- o Delete the entry whose name is specified
- o Add an entry containing the combined statistics from the selected program executions and assign it a specified name

The report-generation specification may request any combination of the following:

- o A Detailed Execution Report on the statistics in the execution statistics file
- o A Processor Summary Report for the selected program executions
- o A Program Summary Report for the selected program executions

Each specification shall be optional. Their formats shall be determined during detailed design of the system.

S3.3.2 Outputs

The output of the system shall consist of the following:

- o The instrumented source code
- o A report on the user inputs to Phase 1
- o A report on the user inputs to Phase 2
- o A Detailed Execution Report, if requested
- o A Processor Summary Report, if requested
- o A Program Summary Report, if requested

The instrumented source code shall be directed to a disk file. Each logical record of this file shall consist of an 80-character image. The reports shall be directed to a file that may be output on a line printer or a terminal. The formats of these reports shall be as indicated in Figures S3 through S7.

S3.4 <u>Data Characteristics</u>

The SMITE program to be processed may reside in a system data base. In addition, the following files shall be created and/or used by the system:

```
EXECUTION MONITOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT:

PROCESSORS MONITORED
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
          .
          .
          .

MONITORING METHOD:  XXXXXXXXX
```

Figure S3. Format of the Report on Phase 1 User Inputs

```
EXECUTION MONITOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT:

REPORTS REQUESTED:  XXXXXXXXXX  XXXXXXXXXX  XXXXXXXXXX

EXECUTION STATISTICS MERGED INTO XXXX:
         XXXX  XXXX  XXXX   .   .   .

EXECUTION STATISTICS DELETED:  XXXX
```

Figure S4. Format of the Report on Phase 2 User Inputs

```
                  PROCESSOR XXXXXXXXXX

SEG#    LINE#       SOURCE STATEMENTS                                                                      EXECUTION
XXX     XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
        XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
        XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
        XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
XXX                                                                                                                TRUE  XXXXX
XXX                                                                                                                FALSE XXXXX
XXX     XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
        XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
XXX                                                                                                                CASE1 XXXXX
XXX                                                                                                                CASE2 XXXXX
XXX                                                                                                                CASE3 XXXXX
XXX     XXX         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX     XXXXXX
 .       .                          .                                                                        .
 .       .                          .                                                                        .
 .       .                          .                                                                        .
```

Figure S5. Format of the Detailed Execution Report

```
                       SUMMARY FOR PROCESSOR XXXXXXXXXX

                NUMBER OF SOURCE STATEMENTS        = XXXXX
                NUMBER OF SEGMENTS                 = XXXXX

                EXECUTABLE SOURCE STATEMENTS       = XXXXX = XXX.X%
                NON-EXECUTABLE SOURCE STATEMENTS   = XXXXX = XXX.X%

                EXECUTABLE STATEMENTS EXECUTED     = XXXXX = XXX.X%
                SEGMENTS EXECUTED                  = XXXXX = XXX.X%

SEG#   LINES     TOTAL    RUN XX    RUN XX    RUN XX    RUN XX    RUN XX

XXX    XXX-XXX   XXXXX    XXXXX     XXXXX     XXXXX     XXXXX     XXXXX
XXX    XXX-XXX   XXXXX    XXXXX     XXXXX     XXXXX     XXXXX     XXXXX
XXX    XXX-XXX   XXXXX    XXXXX     XXXXX     XXXXX     XXXXX     XXXXX
XXX    XXX-XXX   XXXXX    XXXXX     XXXXX     XXXXX     XXXXX     XXXXX
 .        .        .        .         .         .         .         .
 .        .        .        .         .         .         .         .
 .        .        .        .         .         .         .         .
```

Figure S6. Format of the Processor Summary Report

```
                          SUMMARY OF EXECUTION COVERAGE

              NUMBER OF SOURCE STATEMENTS         = XXXXX
              NUMBER OF SEGMENTS                  = XXXXX

              EXECUTABLE SOURCE STATEMENTS        = XXXXX = XXX.X%
              NON-EXECUTABLE SOURCE STATEMENTS    = XXXXX = XXX.X%

              EXECUTABLE STATEMENTS               = XXXXX = XXX.X%
              SEGMENTS EXECUTED                   = XXXXX = XXX.X%

                         TOTAL     SEGMENTS             EXECUTABLE
   PROCESSOR            SEGMENTS   EXECUTED   PERCENT   STATEMENTS   EXECUTED   PERCENT

   XXXXXXXXXX             XXX        XXX       XXX.X       XXX          XXX      XXX.X
   XXXXXXXXXX             XXX        XXX       XXX.X       XXX          XXX      XXX.X
   XXXXXXXXXX             XXX        XXX       XXX.X       XXX          XXX      XXX.X
   XXXXXXXXXX             XXX        XXX       XXX.X       XXX          XXX      XXX.X
       .                   .          .          .          .            .         .
       .                   .          .          .          .            .         .
       .                   .          .          .          .            .         .
```

Figure S7. Format of the Program Summary Report

- A support file of statistics-collecting routines to be developed as part of the system and used in instrumenting the SMITE source code

- A segmented listing file for each instrumented program version

- A file of execution statistics for each execution of instrumented code

- An execution history file to be created and maintained by Phase 2 of the system

Each of these files shall be stored on disk. The support file will require approximately 5,000 words of storage and is to be retained as long as the Execution Monitor is being used. It shall contain all of the statistics-collecting routines that will be required to instrument SMITE source code. A segmented listing file will be produced each time Phase 1 is executed. The size of each file will depend upon the SMITE program being processed. Each file is to be retained as long as it may be needed as input to Phase 2 to generate a Detailed Execution Report for the SMITE program version. A file of execution statistics will be produced each time that instrumented source code is executed. The size of each file will depend upon the SMITE program being processed. Each file is to be retained as long as it may be needed to generate any of the Phase 2 execution reports. There shall be a single history file for the system. Its size will depend upon the number of execution statistics stored. It is to be retained as long as the Execution Monitor is being used.

S3.5 Failure Contingencies

Not applicable.

S4. ENVIRONMENT

S4.1 Equipment Environment

The Execution Monitor will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 100K words of main memory to operate. The system will be stored on disk, as will the SMITE program to be analyzed, the support file, the segmented listing files, the instrumented source code, the files of execution statistics, and the execution history file. The user inputs may be entered via card reader or terminal. The system-generated reports will be directed to a file that may be output on a line printer or a terminal. No new equipment will be required.

S4.2 Support Software Environment

See Appendix G.

S4.3 Interfaces

The system will not interface with any other system.

S4.4 Security

The system will have no classified components.

S5. COST FACTORS

Development of the Execution Monitor will require approximately 25 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the system.

S6. DEVELOPMENTAL PLAN

The Execution Monitor is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Execution Monitor is given in Figure S8.

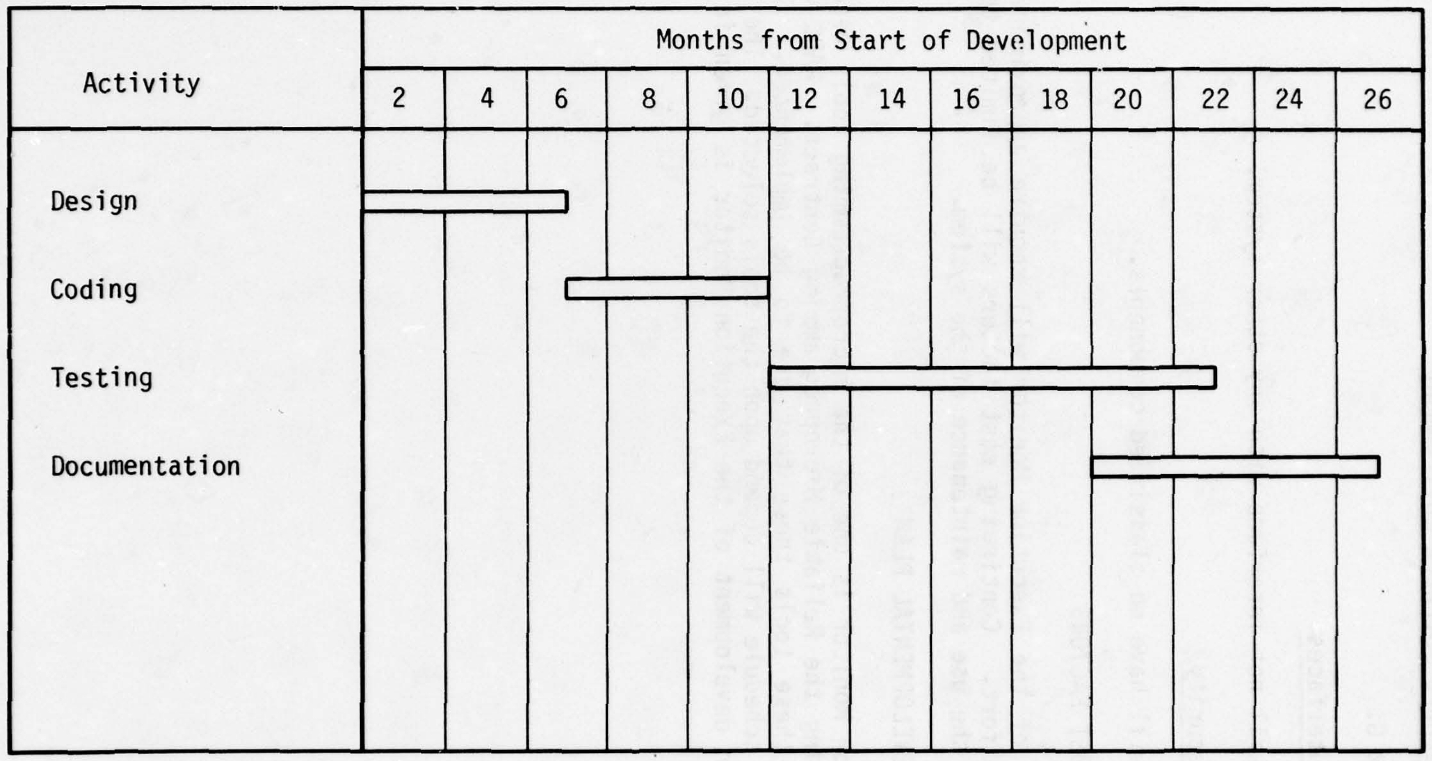


Figure S8. Development Schedule for the Execution Monitor

APPENDIX T
SYSTEM SPECIFICATION FOR THE EXECUTION MONITOR

T1. GENERAL

T1.1 Purpose of the System Specification

See Appendix H.

T1.2 Project References

See Appendix H.

T2. SUMMARY OF REQUIREMENTS

T2.1 System Description

The Execution Monitor shall provide an automated means of determining the execution characteristics of a program written in SMITE. The system shall operate in two independent phases. Phase 1 shall process and report on user inputs, analyze the source code and divide it into logical code segments, generate a file containing a listing of the segmented code, and instrument the source code with statistics-collecting instructions. At the conclusion of Phase 1, the instrumented source code may be compiled and then executed any number of times, generating for each execution a file of statistics describing the execution history of each code segment. Phase 2 of the system shall process and report on user inputs, record a set of execution statistics in a history file, if requested, and prepare a series of reports on selected sets of execution statistics.

The system shall consist of two computer programs. Figure T1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

T2.2 System Functions

The Execution Monitor shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o Source Code Instrumentation
- o Execution Statistics Processing

The following paragraphs identify the requirements to be satisfied by each of these functions.

T2.2.1 Source Code Instrumentation Function

The Source Code Instrumentation function shall accept as input a set of user-prepared inputs and the source code of a program written in SMITE. The user-prepared inputs shall specify the following:

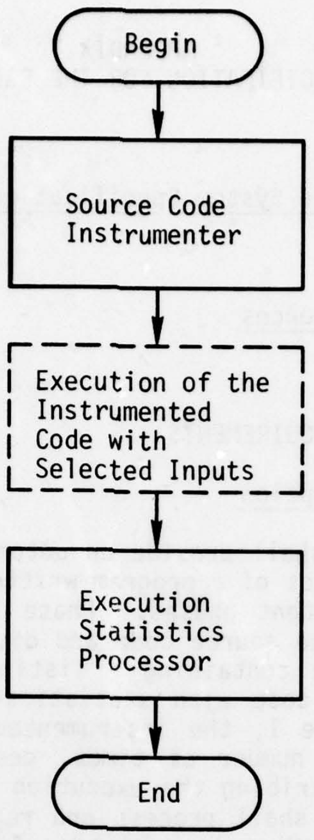

Begin
Source Code Instrumenter
Execution of the Instrumented Code with Selected Inputs
Execution Statistics Processor
End

Figure T1. Relationship of the Components of the Execution Monitor

- o The processors to be monitored
- o Whether the monitoring is to count the number of times each code segment is executed or simply identify those segments that have been executed at least once

Default options shall be provided for each specification. These options shall specify the following:

- o All processors in the program are to be monitored
- o The monitoring is to count the number of times each segment is executed

The function shall interpret the inputs, check them for errors in format and content, store the specified options, and prepare a report that lists the inputs and identifies any errors detected in them. If errors are detected, the function shall request and process corrections. When the inputs are correct, the function shall process the SMITE source code.

In processing the source code, the function shall perform the following tasks:

- o Divide the selected processors into logical code segments that can be used to monitor the execution of the processor
- o Number the code segments and generate a segmented listing file of the selected processors, i.e., a file containing a listing of the source code with each segment appropriately numbered
- o Generate an instrumented version of the source code, i.e., a version that has been modified by addition of the following:
  - - Special statements in each code segment to permit the execution history of that segment to be monitored
  - - Appropriate declarations and initializing code for all monitoring variables
  - - Statistics-collecting routines taken from a support file to be developed in conjunction with the system
  - - Special instructions to cause the execution statistics to be written into a file

To limit execution time and core requirements, the function shall expect the selected processors to be syntactically correct and to contain no more than 2,000 code segments. If any source statement cannot be interpreted, the function shall indicate this problem in the report on user inputs and shall proceed to the next source statement. If more than 2,000 code segments are encountered, the function shall indicate this problem in the report on user inputs, instrument 2,000 of the segments, and leave the remainder uninstrumented.

The source code instrumentation shall be sufficient to cause a file of execution statistics to be generated during each execution of the instrumented program. These statistics shall either identify all code segments that have been executed in the selected processors or indicate the number of times each segment has been executed, depending upon the user specifications to the function.

T2.2.2 Execution Statistics Processing Function

The Execution Statistics Processing function shall accept the following inputs:

- o A set of user-prepared inputs
- o The segmented listing file for an instrumented SMITE program
- o The file of execution statistics generated during one execution of the SMITE program
- o A history file of previously recorded execution statistics

The user-prepared inputs shall specify the following:

- o The manner in which the history file is to be updated, if at all
- o The types of reports that are to be generated (Detailed Execution Report, Processor Summary Report, and/or Program Summary Report)
- o The names of up to 10 program executions whose statistics are to be used in the updating and reporting tasks (the execution covered by the execution statistics file shall be one of these)

Default values shall be provided for each specification. These values shall specify the following:

- o No action is to be taken on the history file
- o All three reports are to be generated
- o Only the statistics from the execution statistics file are to be used for the reports

The function shall interpret the inputs, check them for errors in format and content, store the specified options, and prepare a report that lists the inputs and identifies any errors detected in them. If errors are detected, the function shall request and process corrections. When the inputs are correct, the user requests shall be carried out.

The function shall be capable of updating the history file in either of the following ways, depending upon user specification:

- o Deleting a specified entry from the file
- o Adding an entry containing the combined statistics from the selected program executions and assigning it a user-specified name

If the second option is selected and more than one execution has been selected, the method of combining the statistics shall be as follows:

- o If the statistics indicate the number of times each code segment was executed, the value for each segment in the combined entry shall be the sum of the values associated with that segment in the selected executions.
- o If the statistics simply indicate whether each code segment was executed at least once, the value for each segment in the combined entry shall indicate whether the segment was executed during any of the selected executions.

If the user selects executions whose statistics are not all of one type, the function shall output an error message in the report on user inputs and shall permit correction of the request. The function shall permit up to 999 entries to exist in the history file at one time. If the user attempts to exceed this limit or if the name he provides for a new entry is a duplicate of an existing entry, the function shall output an error message in the report on user inputs and shall permit correction of the request.

Besides updating the history file, the function shall be capable of generating the following reports:

- o A Detailed Execution Report
- o A Processor Summary Report
- o A Program Summary Report

Each of these reports shall be optional. The Detailed Execution Report shall present the statistics from the execution statistics file. It shall contain a source listing of the monitored processors, identifying each code segment and indicating the execution history of that segment (i.e., the number of times the segment was executed or the fact that it was executed at least once). The Processor Summary Report shall indicate, for each monitored processor, the execution history of each code segment during the selected program executions and the percentage of statements and branches executed. The Program Summary Report shall indicate the percentage of all statements and segments executed in each monitored processor during the selected executions and the overall percentages for the monitored portion of the program.

T2.3 Accuracy and Validity

The percentages presented in the Processor Summary and Program Summary Reports shall be given to the nearest tenth of a percent.

T2.4 Timing

There are no timing requirements on the system.

T2.5 Flexibility

The system shall be designed in such a way that the limit on the number of branches in the SMITE program and on the number of entries in the history file can be easily modified.

T3. ENVIRONMENT

T3.1 Equipment Environment

See Appendix H for equipment description.

The Execution Monitor will require approximately 100K words of main memory. The system will be stored on disk, requiring approximately 60K words of storage for the source code, object code, and load module. Also

stored on disk will be the SMITE program to be analyzed, the support file, the segmented listing files, the instrumented source code, the files of execution statistics, and the execution history file. The user inputs may be entered via card reader or terminal. The system-generated reports will be directed to a file that may be output on a line printer or a terminal. No new equipment will be required.

T3.2 Support Software Environment

See Appendix H.

T3.3 Interfaces

The system shall not interface with any other system.

T3.4 Security

The system shall have no classified components.

T3.5 Controls

No controls shall be established by the system.

T4. DESIGN DATA

T4.1 System Logical Flow

The Execution Monitor shall be a system consisting of two computer programs:

- o The Source Code Instrumenter
- o The Execution Statistics Processor

The Source Code Instrumenter shall constitute Phase 1 of the system; the Execution Statistics Processor shall constitute Phase 2. For a given set of processors in a SMITE program version, Phase 1 is meant to be executed once, Phase 2 any number of times. Once the SMITE program has been instrumented by Phase 1, it may be compiled and then executed repeatedly using different inputs to exercise different program paths. Phase 2 may be invoked as often as desired to record and report on the resulting execution statistics. Figure T2 illustrates the logical flow of the system and indicates the inputs and outputs of each step of system operation, including the data base files created and used at each step.

T4.2 Program Descriptions

The Execution Monitor shall be comprised of the two computer programs identified in Section T4.1. Design data for each program follow.

T4.2.1 Source Code Instrumenter Program

The Source Code Instrumenter shall fulfill the requirements specified in Section T2.2.1. Additional design data follow.

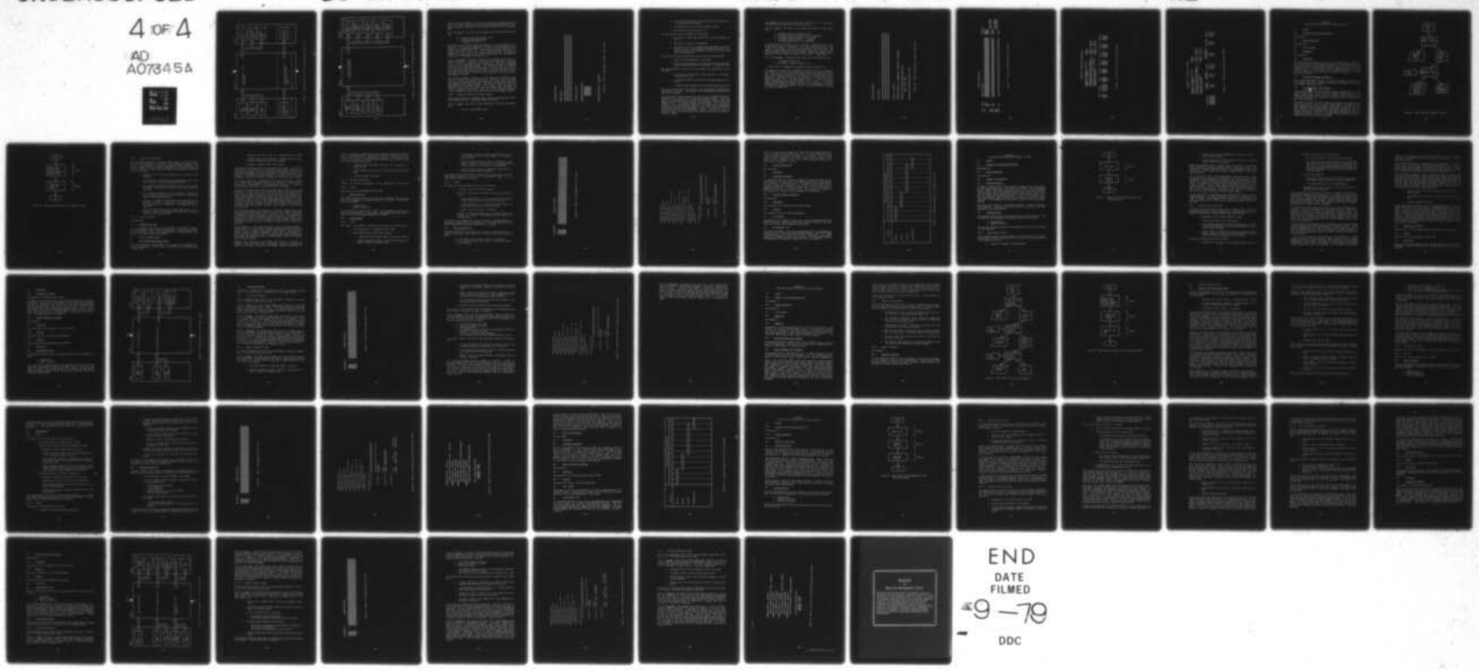
T4.2.1.1 Inputs: The inputs to the program shall consist of a set of user-prepared inputs and the source code for a program written in SMITE. The user inputs may specify the following:

- o The processors to be monitored
- o Whether the monitoring is to count the number of times each code segment has been executed or simply identify those segments that have been executed at least once

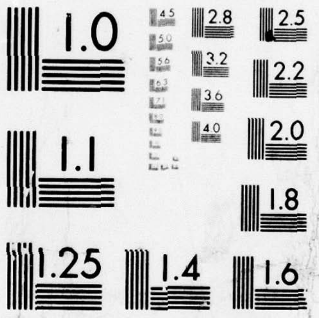
Each specification shall be optional. The program shall accept them via card reader or terminal. Their formats shall be determined during detailed design of the system.

AD-A073 454 LOGICON INC SAN PEDRO CALIF F/G 9/2
RELIABLE MICROPROGRAMMING.(U)
JUL 79 J RADATZ, J LAUB, G HAMACHI F30602-78-C-0079
UNCLASSIFIED DS-R79001 RADC-TR-79-173 NL

4 OF 4
AD
A073454



END
DATE
FILMED
9-79
DDC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

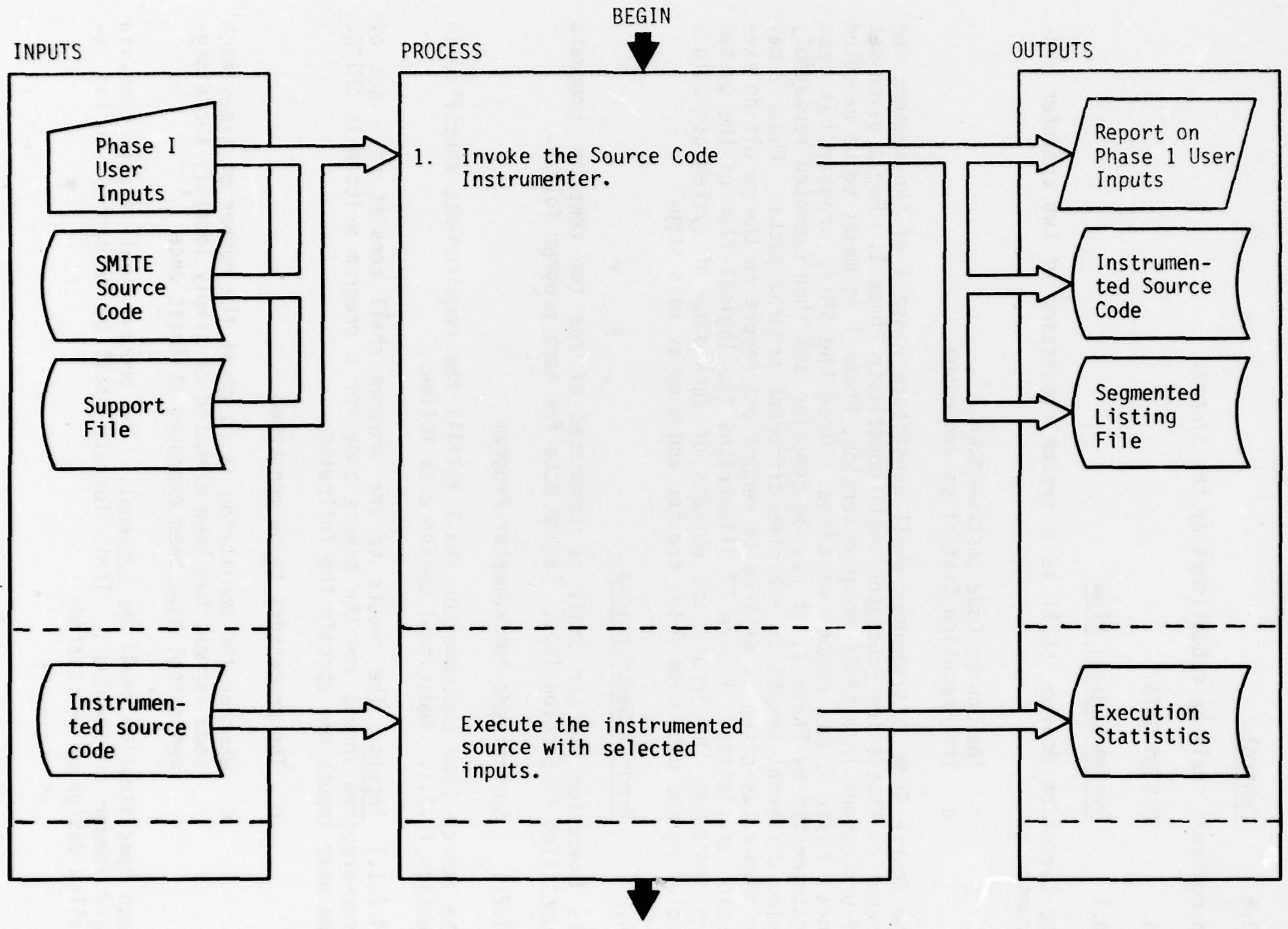


Figure T2. System Logical Flow for the Execution Monitor (Page 1 of 2)

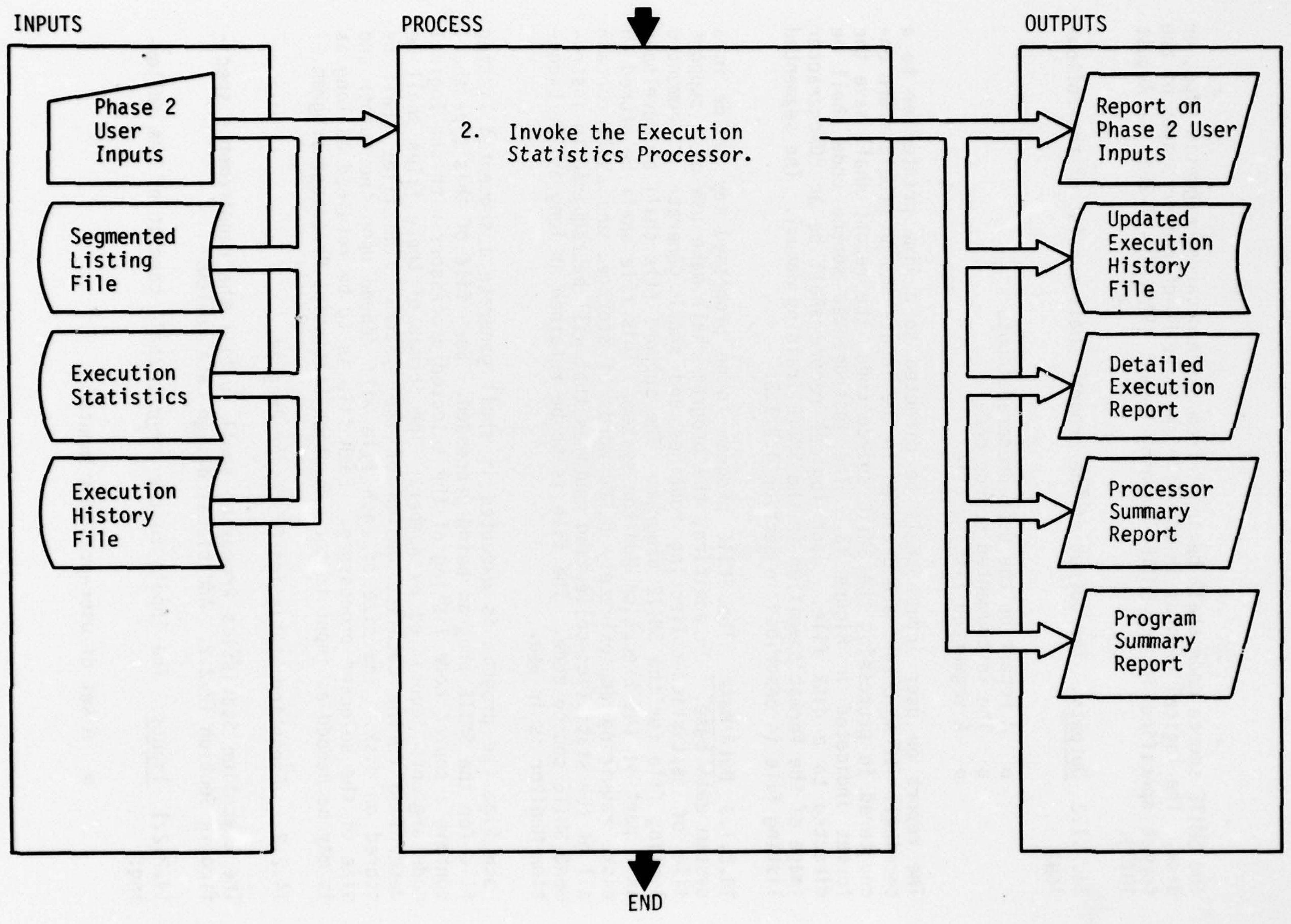


Figure T2. System Logical Flow for the Execution Monitor (Page 2 of 2)

The SMITE source code shall be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

T4.2.1.2 Outputs: The output of the program shall consist of the following:

- o A report on the user-prepared inputs
- o The instrumented source code
- o A segmented listing file

The report on user inputs shall be directed to a line printer or to a terminal. If no errors are detected in the inputs and no problems are encountered in processing the SMITE source code, the report shall have the format indicated in Figure T3. The instrumented source code shall be directed to a disk file. Each logical record shall be an 80-character image of the format specified in the SMITE Training Manual. The segmented listing file is described in Section 4.2.1.3.

T4.2.1.3 Data Base: The SMITE program to be processed may reside in a system data base. In addition, the program shall make use of a support file of statistics-collecting routines and shall generate a segmented listing file for the SMITE program. The support file shall be developed as a part of the Execution Monitor system. This file shall be stored on disk, requiring approximately 5,000 words of storage, and shall contain all of the statistics-collecting routines that will be required to instrument SMITE source code. The file is to be retained as long as the Execution Monitor is in use.

Each time the program is executed it shall generate a segmented listing file for the SMITE program being processed. Each file of this type shall contain a source code listing of the selected processors with the logical code segments identified by number. The format of these files shall be determined during detailed design of the system. The files shall be stored on disk. The size of each file will depend upon the number and size of the selected processors. Each file is to be retained as long as it may be needed as input to the Execution Statistics Processor program.

T4.2.2 Execution Statistics Processor Program

The Execution Statistics Processor shall fulfill the requirements specified in Section T2.2.2. Additional design data follow.

T4.2.2.1 Inputs: The inputs to the program shall consist of the following:

- o A set of user-prepared inputs

```
EXECUTION MONITOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT:

PROCESSORS MONITORED
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
      XXXXXXXXXX
          .
          .
          .

MONITORING METHOD:  XXXXXXXXX
```

Figure T3. Format of the Report on Phase 1 User Inputs

- A file of execution statistics generated during execution of an instrumented SMITE program
- The segmented listing file for the SMITE program
- The system's execution history file

The user-prepared inputs may specify the following:

- The manner in which the history file is to be updated, if at all
- The types of reports to be generated
- The names of up to 10 program executions whose statistics are to be used in the updating and reporting tasks. (The execution covered by the execution statistics file shall be one of those selected)

The history-file-update specification may be either of the following:

- Delete the entry whose name is specified
- Add an entry containing the combined statistics from the selected program executions and assign it a specified name

The report-generation specification may request any combination of the following:

- A Detailed Execution Report on the statistics in the execution statistics file
- A Processor Summary Report for the selected program executions
- A Program Summary Report for the selected program executions

Each specification shall be optional. The inputs shall be entered via card reader or terminal. Their format shall be determined during detailed design of the system.

The second input to the program shall be a file of execution statistics generated during execution of an instrumented SMITE program. This file shall contain statistics that indicate the execution history of each code segment in each instrumented processor of the program. The file shall be stored on disk. Its size will depend upon the number, size, and logical complexity of the instrumented processors. Its format shall be determined during detailed design of the system. There will be one such file for each execution of an instrumented SMITE program. Each file is to be retained as long as it may be required as input to the Execution Statistics Processor program

The segmented listing file is described in Section T4.2.1.3. The execution history file is described in Section T4.2.2.3.

T4.2.2.2 Outputs: The output of the program shall consist of the following:

- o A report on the user-prepared inputs
- o An updated execution history file, if requested
- o A Detailed Execution Report, if requested
- o A Processor Summary Report, if requested
- o A Program Summary Report, if requested

The updated execution history file is described in Section T4.2.3.3. All of the reports shall be directed to a file that may be output on a line printer or a terminal. If no errors are detected in the user inputs, the report on these inputs shall have the format indicated in Figure T4. The Detailed Execution Report, Processor Summary Report, and Program Summary Report shall be as indicated in Figure T5, T6, and T7, respectively.

T4.2.2.3 Data Base: The program shall make use of the following files:

- o A segmented listing file
- o The system's execution history file

It shall create, if requested, an updated execution history file.

The segmented listing file is described in Section T4.2.1.3. The execution history file shall contain execution statistics recorded by the system. Each entry in the file shall contain either the statistics from a single SMITE program execution or the combined statistics from up to 10 executions. Each entry shall have a unique name by which the user and the system may refer to it. The file shall be stored on disk. Its format shall be determined during detailed design of the system. Its size will depend upon the number of entries and the size of each entry. A maximum of 999 entries shall be permitted. The file is to be retained as long as the system is being used.

```
EXECUTION MONITOR

USER INPUTS:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

OPTIONS IN EFFECT:

REPORTS REQUESTED:  XXXXXXXXXX  XXXXXXXXXX  XXXXXXXXXX

EXECUTION STATISTICS MERGED INTO XXXX:
         XXXX  XXXX  XXXX   .   .   .

EXECUTION STATISTICS DELETED:  XXXX
```

Figure T4. Format of the Report on Phase 2 User Inputs

```
                             PROCESSOR XXXXXXXXXX

SEG#   LINE#      SOURCE STATEMENTS                                                                             EXECUTION

XXX    XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
       XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
       XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
       XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
XXX                                                                                                   TRUE  XXXXX
XXX                                                                                                   FALSE XXXXX
XXX    XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
       XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
XXX                                                                                                   CASE1 XXXXX
XXX                                                                                                   CASE2 XXXXX
XXX                                                                                                   CASE3 XXXXX
XXX    XXX        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX    XXXXXX
 .      .                            .                                                           .
 .      .                            .                                                           .
 .      .                            .                                                           .
```

Figure T5. Format of the Detailed Execution Report

```
                        SUMMARY FOR PROCESSOR XXXXXXXXXX

                 NUMBER OF SOURCE STATEMENTS        = XXXXX
                 NUMBER OF SEGMENTS                 = XXXXX

                 EXECUTABLE SOURCE STATEMENTS       = XXXXX = XXX.X%
                 NON-EXECUTABLE SOURCE STATEMENTS   = XXXXX = XXX.X%

                 EXECUTABLE STATEMENTS EXECUTED     = XXXXX = XXX.X%
                 SEGMENTS EXECUTED                  = XXXXX = XXX.X%

SEG#   LINES      TOTAL    RUN XX   RUN XX   RUN XX   RUN XX   RUN XX

XXX    XXX-XXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX
XXX    XXX-XXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX
XXX    XXX-XXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX
XXX    XXX-XXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX    XXXXX
 .        .         .        .        .        .        .        .
 .        .         .        .        .        .        .        .
 .        .         .        .        .        .        .        .
```

Figure T6. Format of the Processor Summary Report

```
                         SUMMARY OF EXECUTION COVERAGE

                 NUMBER OF SOURCE STATEMENTS       = XXXXX
                 NUMBER OF SEGMENTS                = XXXXX

                 EXECUTABLE SOURCE STATEMENTS      = XXXXX = XXX.X%
                 NON-EXECUTABLE SOURCE STATEMENTS  = XXXXX = XXX.X%

                 EXECUTABLE STATEMENTS             = XXXXX = XXX.X%
                 SEGMENTS EXECUTED                 = XXXXX = XXX.X%

                        TOTAL     SEGMENTS              EXECUTABLE
PROCESSOR              SEGMENTS   EXECUTED   PERCENT    STATEMENTS    EXECUTED   PERCENT

XXXXXXXXXX               XXX        XXX       XXX.X        XXX          XXX       XXX.X
XXXXXXXXXX               XXX        XXX       XXX.X        XXX          XXX       XXX.X
XXXXXXXXXX               XXX        XXX       XXX.X        XXX          XXX       XXX.X
XXXXXXXXXX               XXX        XXX       XXX.X        XXX          XXX       XXX.X
    .                     .          .          .           .            .          .
    .                     .          .          .           .            .          .
    .                     .          .          .           .            .          .
```

Figure T7. Format of the Program Summary Report

APPENDIX U
FUNCTIONAL DESCRIPTION FOR THE SYMBOLIC EXECUTOR

U1. GENERAL

U1.1 Purpose of Functional Description

See Appendix G.

U1.2 Project References

See Appendix G.

U2. SYSTEM SUMMARY

U2.1 Background

See Appendix G.

U2.2 Objectives

The purpose of the Symbolic Executor is to provide a symbolic representation of the results of SMITE program execution. These results will identify paths taken during program execution, the input conditions that caused each path to be traversed, and the symbolic values of resulting program outputs. By permitting the user to compare the symbolic results with the associated path condition, the Symbolic Executor will aid in program verification.

U2.3 Existing Methods and Procedures

SMITE program development at RADC is currently performed without the aid of a Symbolic Executor. Program verification is performed with specific values rather than symbolic data.

U2.4 Proposed Methods and Procedures

The proposed method of obtaining symbolic results corresponding to a large number of test cases is for the user to submit a SMITE program to the Symbolic Executor and interact with the system to obtain desired results. The Symbolic Executor will operate in two phases, each of which may be executed independently.

Phase 1 will analyze the SMITE source code, generate a numbered source listing, and build an internal model of the program. Using the internal model together with inputs supplied interactively by the user, Phase 2 will symbolically execute the program and generate intermediate and final results consisting of a description of each path executed, the input conditions that caused each path to be executed, and symbolic values of requested variables. Figure U1 illustrates the data flow for the system. Figure U2 shows its major processing steps.

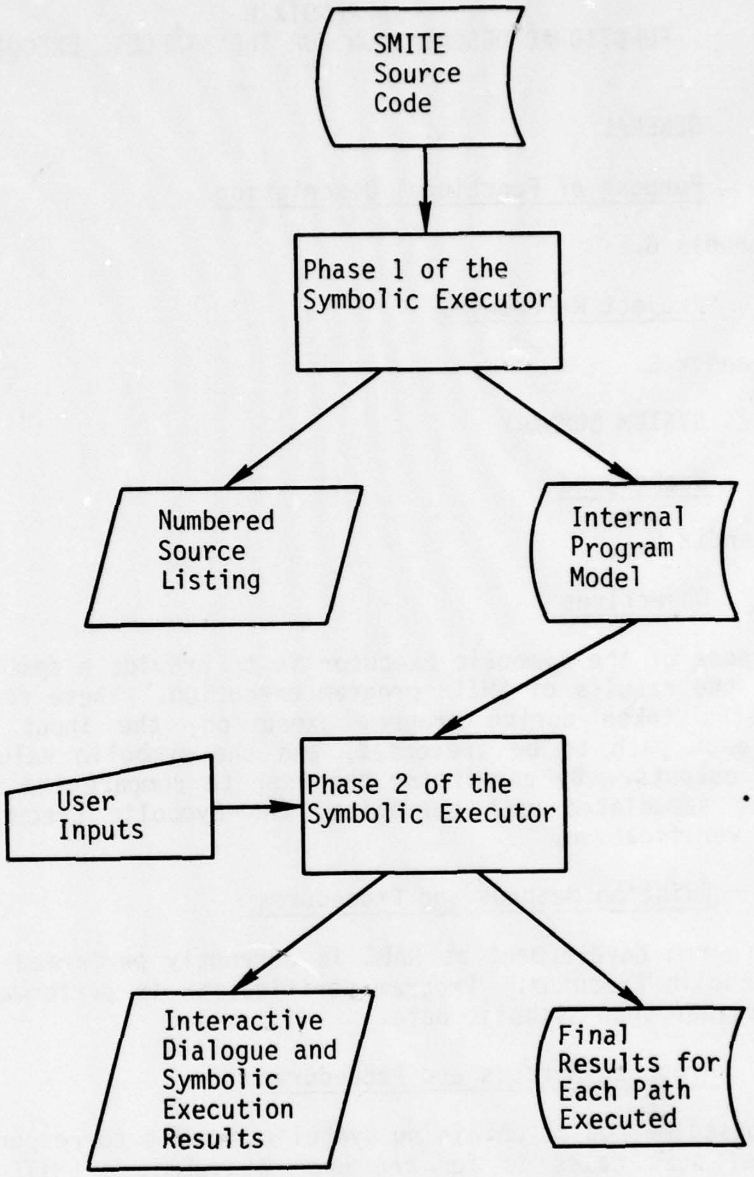

SMITE Source Code
Phase 1 of the Symbolic Executor
Numbered Source Listing
Internal Program Model
User Inputs
Phase 2 of the Symbolic Executor
Interactive Dialogue and Symbolic Execution Results
Final Results for Each Path Executed

Figure U1. Data Flow for the Symbolic Executor

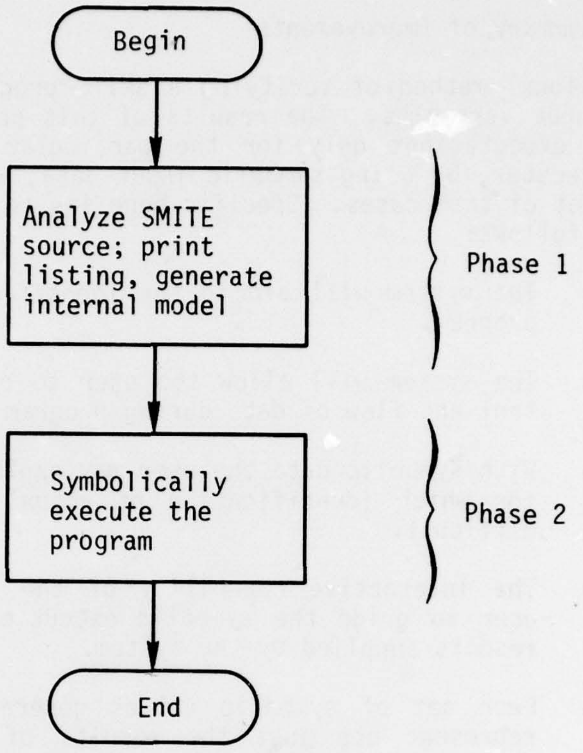

Begin
Analyze SMITE source; print listing, generate internal model
Phase 1
Symbolically execute the program
Phase 2
End

Figure U2. Major Processing Steps in the Symbolic Executor

U2.4.1 Summary of Improvements

The conventional method of verifying a SMITE program is to supply sample data for input variables. The results of this process allow the user to affirm his expectations only for the particular sample selected. The Symbolic Executor, by using symbolic input data, will provide results for an entire set of test cases. Specific benefits to be gained from the system are as follows:

- o The system will aid in the identification of errors in the program.
- o The system will allow the user to observe the flow of control and flow of data during program execution.
- o With symbolic data the user may explore paths in his program for which identification of actual input values would be difficult.
- o The interactive capability of the system will permit the user to guide the symbolic execution based on intermediate results supplied by the system.
- o Each set of symbolic values generated by the system will represent not just the results of a single execution of the program but of a potentially large number of actual executions.
- o Each path condition generated by the system may be used to identify a specific set of data values which will cause the associated path to be executed.

U2.4.2 Summary of Impacts

See Appendix G.

U2.5 <u>Expected Limitations</u>

To limit execution time and core requirements, the Symbolic Executor will be designed to handle a SMITE program that is syntactically correct and that contains no more than 1,000 statements, 1,000 variables, 250 branches, and 10,000 variable occurrences.

U3. DETAILED CHARACTERISTICS

U3.1 <u>Specific Performance Requirements</u>

The Symbolic Executor shall operate in two phases, each designed to be executed independently. Phase 1 shall be responsible for performing the following tasks:

- Analyzing the source code of a program written in SMITE
- Assigning each source statement a sequence number and generating a numbered listing of the program
- Buildng an internal model of the program

If any source statement cannot be interpreted during Phase 1, it shall be identified in the numbered listing and the system shall proceed to the next source statement. If any of the numerical limitations given in Section U2.5 are exceeded during the processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains.

The internal model that is generated shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of each statement's function. It shall be stored on disk to be used as input to Phase 2 of the system.

In Phase 2, the system shall use the internal model to execute the program symbolically. User inputs to this phase shall specify the internal model to be used, variable constraints and initializations, path selection commands, and the symbolic execution results that are to be displayed. Default options shall be provided.The system shall permit the user to submit a complete or partial set of inputs before processing begins. If a partial set of inputs is supplied, the system shall request additional inputs as needed. When such a request is made, the user shall also be permitted to insert other path selection commands, output requests, and processing specifications.

To symbolically execute the program, the system shall proceed along a path that is within the constraints specified by the user. When it encounters a statement that alters the value of a variable, it shall use substitution to update the internal representation of the variable. This substitution shall consist of taking the current symbolic value of each variable in the statement and using this value to replace the variable name. The resulting expression shall be the new symbolic value of the variable being modified.

During processing, if the system encounters a branch or array reference it cannot resolve, it shall output a message requesting further information, permit the user to respond interactively, then proceed. When the system reaches a point at which intermediate results have been requested, it shall output the requested results. In executing the program symbolically the system shall process through routine boundaries and shall correctly handle variable overlays.

Execution shall continue in this manner until the end of the path is reached. The system shall then, if requested, display on a terminal the results for that path, identifying the statements executed along the path, the input values that caused the path to be executed, and/or the symbolic values of program variables. These results shall also be stored in a disk file for later output to a line printer or terminal. After directing the results to a disk file, the system shall permit the user to select one of the following processing options:

- o Process another path which satisfies user constraints, if there are any
- o Cease execution of this class of paths and accept new input data
- o Terminate symbolic execution

U3.1.1 Accuracy and validity

Accuracy and validity requirements are not applicable to the system.

U3.1.2 Timing

There are no timing requirements on the system.

U3.2 <u>System Functions</u>

The Symbolic Executor shall perform two major functions, each of which shall be implemented as a separate program in the system. The functions shall be as follows:

- o Preprocessing
- o Symbolic Execution

The Preprocessing function shall fulfill the requirements described in Section U3.1 for Phase 1 of the system. The Symbolic Execution function shall fulfill the requirements specified for Phase 2.

U3.3 Inputs/Outputs

U3.3.1 Inputs

The inputs to the Symbolic Executor shall consist of the following:

- o The source code for a program written in SMITE
- o User-supplied inputs, which shall include:
  - Designation of the internal model to be used for Phase 2
  - Variable constraints and/or variable initializations in terms of numeric or symbolic values

- Path selection commands indicating the outcome of specified branches and the maximum number of iterations of specified loops
- Output requests specifying program locations at which symbolic execution results are to be displayed and the results that are to be displayed at each location
- Specific numeric values for unresolved array indices as they are encountered by the system

The source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. The user inputs shall be entered via terminal.

U3.3.2 Outputs

The output of the system shall consist of the following:

o A numbered listing of the SMITE program

o A disk file contaning final results for each path, consisting of:

  - A path description, i.e., the sequence of statement numbers encountered during traversal of the path
  - A path condition, i.e., conditions on the input variables which caused the path to be executed
  - Symbolic values of requested variables

o Interactive dialogue between user and system, which shall include, as requested, intermediate and final results as outlined above

The format of the numbered listing and interactive diaglogue shall be as indicated in Figures U3 and U4, respectively. The format of the disk file shall be determined during detailed design of the system.

U3.4 Data Characteristics

The SMITE program to be processed may reside in a system data base. In addition, the system shall create the following files to be stored on disk:

o A file containing the internal model of the program

o A file containing the final results for each path symbolically executed

```
STATEMENT #          SOURCE STATEMENT

  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    .                   .
    .                   .
    .                   .
```

Figure U3. Format of the Numbered Source Listing

```
COMMAND ACCEPTED:  XXXXXXXX
    INTERNAL MODEL USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE CONSTRAINT:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE INITIALIZED:  XXXXXXXX = XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    BRANCH OUTCOME SPECIFIED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    NUMBER OF ITERATIONS OF LOOP AT XXXX:  XXXX
COMMAND ACCEPTED:  XXXXXXXX
    LOCATION AT WHICH TO DISPLAY RESULTS:  XXXX
    RESULTS TO BE DISPLAYED:  XXXXXXXX

VALUE REQUIRED FOR UNRESOLVED ARRAY INDEX XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    ARRAY INDEX XXXXXXXX = XXXX
PATH COMPLETED

COMMAND ACCEPTED:  XXXXXXXX
        PATH DESCRIPTION:  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                                   .
                                   .
                                   .
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

        PATH CONDITION:    XXXXXXXXXX &
                           XXXXXXXXXX &
                               .
                               .
                               .
                           XXXXXXXXXX

        SYMBOLIC VALUES:   XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                           XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                                   .
                                   .
                                   .
```

Figure U4. Format of the Symbolic Execution Interactive Dialogue

The size of the internal model will depend upon the program being analyzed, but should not exceed 50K words. It is to be retained as long as it may be needed for executions of Phase 2 of the system. The size of the second file will depend upon the SMITE program being analyzed and the number of paths traversed. This file is to be retained as long as its results are of interest to the user.

U3.5 Failure Contingencies

Not applicable.

U4. ENVIRONMENT

U4.1 Equipment Environment

The Symbolic Executor will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 200K words of main memory. The system will be stored on disk. The program to be analyzed may be stored on punched cards, magnetic tape, or disk. The user inputs will be entered via terminal. The internal model and the file of final symbolic execution results will be stored on disk. The interactive dialogue will be directed to a terminal. No new equipment will be required.

U4.2 Support Software Environment

See Appendix G.

U4.3 Interfaces

The system will not interface with any other system.

U4.4 Security

The system will have no classified components.

U5. COST FACTORS

Development of the Symbolic Executor will require approximately 40 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the system.

U6. DEVELOPMENTAL PLAN

The Symbolic Executor is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall schedule will depend upon the tools selected. The recommended schedule for development of the Symbolic Executor is given in Figure U5.

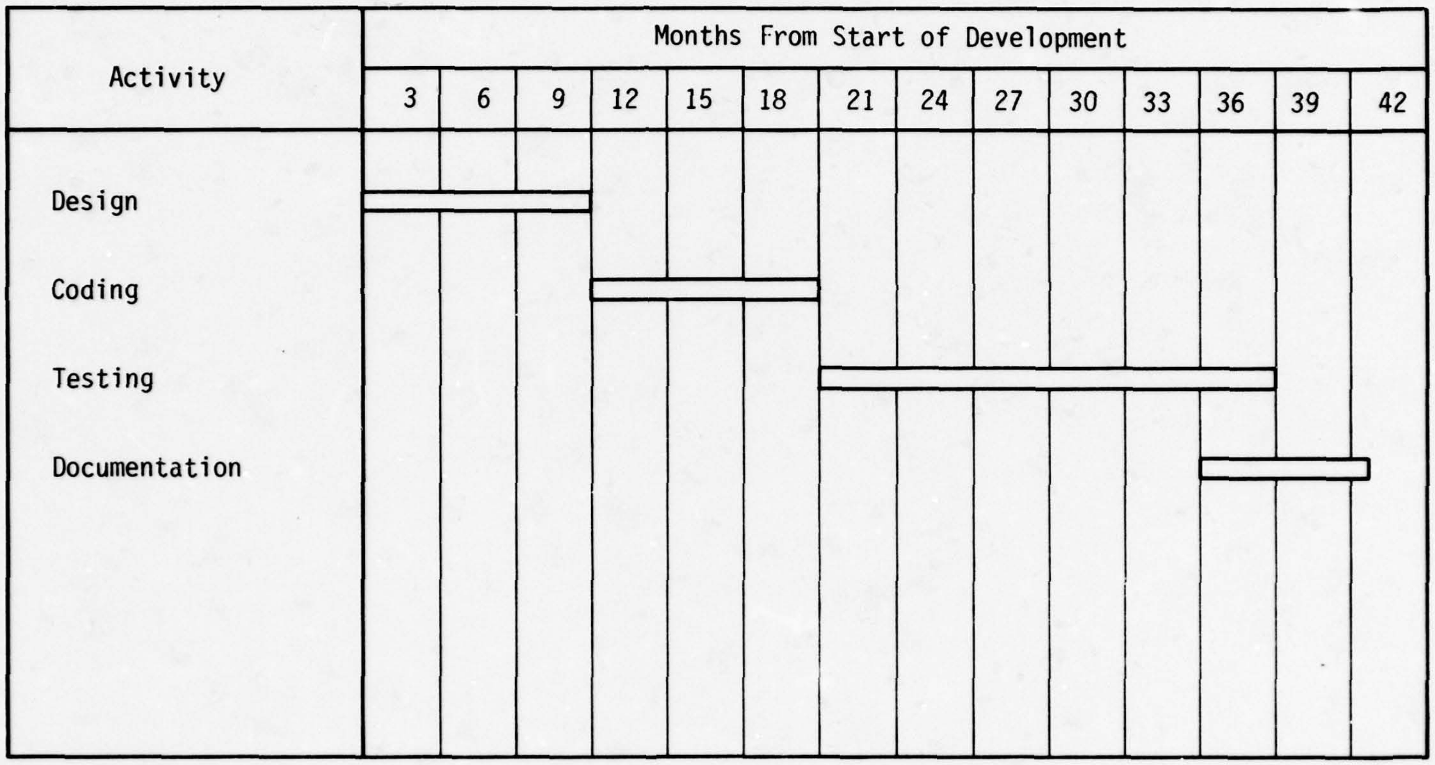


Figure U5. Development Schedule for the Symbolic Executor

APPENDIX V
SYSTEM SPECIFICATION FOR THE SYMBOLIC EXECUTOR

V1. GENERAL

V1.1 Purpose of the System Specification

See Appendix H.

V1.2 Project References

See Appendix H.

V2. SUMMARY OF REQUIREMENTS

V2.1 System Description

The Symbolic Executor shall provide symbolic representation of the results of SMITE program execution. The system shall operate in two independent phases. Phase 1 shall analyze the SMITE source code, generate a numbered source listing, and build an internal model of the program. Phase 2 shall use the internal model together with inputs supplied interactively by the user to execute the program symbolically. The results of symbolic execution shall identify each path executed, the input conditions that caused each path to be traversed, and the symbolic values of resulting program outputs.

The system shall consist of two computer programs. Figure V1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

V2.2 System Functions

The Symbolic Executor shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o Preprocessing
- o Symbolic Execution

The following paragraphs identify the requirements to be satisfied by each of these functions.

V2.2.1 Preprocessing Function

The Preprocessing function shall accept as input the source code of a program written in SMITE. The function shall process each source statement of the program as follows:

- o *Assign the statement a sequence number*

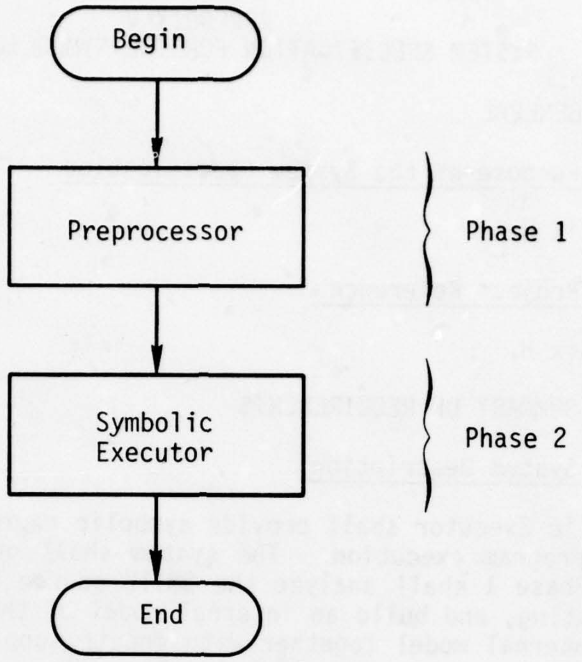


Figure V1. Relationship of the Components of the Symbolic Executor

- o Output a copy of the statement and its sequence number in a listing of the program

- o Analyze the content of the statement and use it to build an internal model of the program

Source code comments shall be copied into the program listing without being analyzed or assigned a sequence number. The internal model shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of the statement's function.

To limit execution time and core requirements, the function shall expect the program to be syntactically correct and to have no more than 1,000 statements, 1,000 variables, 250 branches, and 10,000 variable occurrences. If any source statement cannot be interpreted, the function shall report this problem in the program listing and shall continue to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, then continue to analyze and list each source statement, storing all relevant information for which space remains.

At the conclusion of preprocessing, the function shall output a problem summary message if any source statements could not be interpreted or if space limitations were exceeded in the processing. The internal model shall be stored on disk to be used as input to the Symbolic Execution function.

V2.2.2 Symbolic Execution Function

The Symbolic Execution function shall use the internal model of the SMITE program to execute the program symbolically. User-supplied inputs to be supplied before symbolic execution begins shall be as follows:

- o Designation of the internal model to be used

- o Variable constraints and initializations

- o Path selection commands indicating the branches to be taken at specified branch points and the maximum number of iterations of specified loops

- o Output requests specifying the program locations at which symbolic execution results are to be displayed and the results that are to be displayed at each location

Provision for defaults shall be as follows:

- o There shall be no default for the internal model to be used.

- Variable constraints and initializations:
  - No variable constraints shall be defined by default.
  - The symbolic value of each variable shall be undefined unless the user explicitly assigns it a numeric or symbolic value. For all remaining undefined variables, the user shall have the option to specify with a single command that the symbolic value for each is to be the name of the variable.
- Path selection commands:
  - The function shall request user input at any branch point whose outcome has not been previously specified.
  - All unspecified loops shall be executed once.
- Intermediate and final results for each path shall not be displayed unless requested.

The function shall permit the user to submit a complete or partial set of inputs before symbolic execution begins. If a partial set of inputs is supplied, the function shall request additional inputs as needed. When such a request is made, the user shall also be permitted to insert other path selection commands, output requests, requests that the function cease execution of this class of paths and accept new input data, or requests that the function cease execution altogether.

To execute the program symbolically, the function shall proceed along a path that is within the constraints specified by the user. When it encounters a statement that alters the value of a variable, it shall use substitution to update the internal representation of the variable. This substitution shall consist of taking the current symbolic value of each variable in the statement and using this value to replace the variable name. The resulting expression shall be the new symbolic value of the variable being modified.

In executing the program symbolically, the function shall process through routine boundaries and shall correctly handle variable overlays. If the function encounters an undefined variable, it shall permit the user to initialize the variable or to leave it undefined. If the user chooses to leave the variable undefined, the function shall consider the current path to be terminated.

If the function encounters an array reference it cannot resolve, it shall request a numeric value for the array index, permit the user to respond interactively, then proceed. When the function reaches a point at which intermediate results have been requested, it shall output the requested results, permit user input, then proceed. The intermediate results may identify the statements executed thus far on the path, the input conditions that have caused the path to be executed, and symbolic values for requested variables.

If the function encounters a branch point it cannot resolve, it shall permit the user to specify one or more branches to be executed. If the user selects more than one branch, the system shall note the order in which these branches were selected, preserve the path condition to this branch point, and continue processing, using the first branch selection.

When the function completes a path, it shall, if requested, display the final results for that path. These results shall identify the statements executed along the path, the input conditions that caused the path to be executed, and/or the symbolic values of requested program variables. Unless the path has been terminated prematurely because of an undefined variable, these path results shall be stored on disk for later output to a line printer or terminal. The function then shall permit the user to select one of the following processing options:

- o Process another path which satisfies user constraints, if there are any
- o Cease execution of this class of paths and accept new input data
- o Terminate symbolic execution

If the user selects the first option, the system shall return to the most recent branch point for which there is a selected branch yet to be executed. If such a branch point exists, the function shall request input specifying whether the user wishes to proceed with the next branch selection. If the user's reply is affirmative, the function shall execute this branch. If the reply is negative, the function shall offer another branch selection. When no branch point exists at which a selected branch remains to be executed, the function shall cease execution of this class of paths. The user shall then be permitted to submit a new set of inputs or to terminate symbolic execution.

V2.3 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

V2.4 Timing

There are no timing requirements on the system.

V2.5 Flexibility

The system shall be designed in such a way that the limit on the number of statements, variables, branches, and variable occurrences can be easily modified.

V3. ENVIRONMENT

V3.1 Equipment Environment

See Appendix H for equipment description.

The Symbolic Executor will require approximately 200K words of main memory to operate. The system will be stored on disk, requiring approximately 75K words of storage for the source code, object code, and load module. The user-prepared inputs will be entered via terminal. The SMITE source code may be input from punched cards, magnetic tape, or disk. The internal model and the file of symbolic execution results will be stored on disk. The other outputs will be directed to a line printer or to a terminal. No new equipment will be required.

V3.2 Support Software Environment

See Appendix H.

V3.3 Interfaces

The system shall not interface with any other system.

V3.4 Security

The system shall have no classified components.

V3.5 Controls

No controls shall be established by the system.

V4. DESIGN DATA

V4.1 System Logical Flow

The Symbolic Executor shall be a system consisting of two computer programs:

- o Preprocessor
- o Symbolic Executor

For a given SMITE program version, the Preprocessor is meant to be executed once, the Symbolic Executor any number of times. Figure V2 indicates the logical flow of the system. The figure also indicates the inputs and outputs of each program, including the data base files created and used by each program.

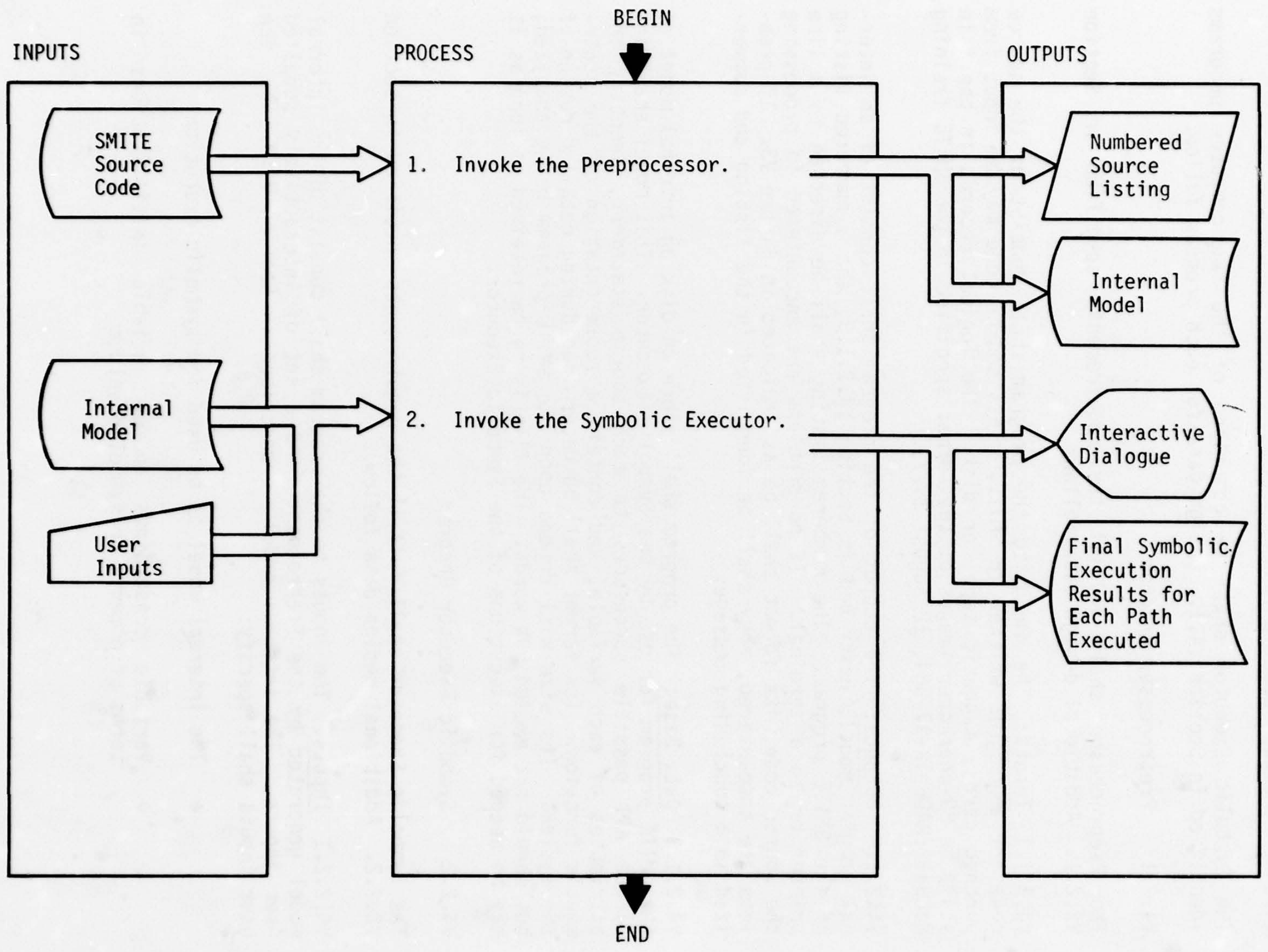


Figure V2. System Logical Flow for the Symbolic Executor

V4.2 Program Descriptions

The Symbolic Executor shall be comprised of the two computer programs identified in Section V4.1. Design data for each program follow.

V4.2.1 Preprocessor Program

The Preprocessor shall fulfill the requirements specified in Section V2.2.1. Additional design data follow.

V4.2.1.1 Inputs: The input to the program shall consist of the source code for a program written in SMITE. This source code may be input from punched cards, magnetic tape, or disk. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

V4.2.1.2 Outputs: The output of the program shall consist of an internal program model, described in Section V4.2.1.3, and a numbered listing of the SMITE program. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, its format shall be as indicated in Figure V3. If problems are encountered, they shall be identified in the listing and summarized in a concluding message.

V4.2.1.3 Data Base: The program shall store on disk an internal model of the SMITE program for use by the Symbolic Executor. This model shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of the statement's function. Its format shall be determined during detailed design of the system. Its size will depend upon the SMITE program being analyzed, but should not exceed 50K words. The file is to be retained as long as it may be needed for executions of the Symbolic Executor.

V4.2.2 Symbolic Executor Program

The Symbolic Executor shall fulfill the requirements specified in Section V2.2.2. Additional design data follow.

V4.2.2.1 Inputs: The inputs to the program shall consist of the internal model generated by the Preprocessor and a set of interactively supplied user inputs. The internal model is described in Section V4.2.1.3. The user inputs shall specify:

- o The internal model to be used for symbolic execution
- o Variable constraints and/or variable initializations in terms of numeric or symbolic values

```
STATEMENT #           SOURCE STATEMENT

   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     .                      .
     .                      .
     .                      .
```

Figure V3. Format of the Numbered Source Listing

- o Path selection commands indicating the outcome of specified branches and the maximum number of iterations of specified loops

- o Output requests specifying the program locations at which symbolic execution results are to be displayed and the results that are to be displayed at each location

- o Specific numeric values for unresolved array indices as they are encountered by the function

- o Processing options as required during symbolic execution

The format of the user inputs shall be determined during detailed design of the system. They shall be input via terminal.

V4.2.2.2 Outputs: The output of the program shall consist of interactive dialogue between the user and the system, including symbolic execution results for each path explored. The program may initiate interactive dialogue by requesting the user to specify:

- o The internal model to be used
- o Values for undefined variables
- o Branch selections
- o The symbolic execution results to be displayed at intermediate and final points on a path
- o Values for unresolved array references
- o The processing to be performed upon completion of a path

The symbolic execution results for each path shall consist of the following:

- o A path description, identifying the order of the source statements encountered during execution of the path

- o A path condition, consisting of the set of input conditions that caused the path to be executed

- o Symbolic values of program variables, expressed in terms of the input variables

The interactive dialogue shall be displayed on a terminal. It shall include intermediate and final symbolic execution results for each path as requested. The final symbolic execution results for each fully executed path shall be directed to a disk file for later output to a terminal or line printer. The format of the interactive dialogue and symbolic execution results that are displayed on the terminal shall be as indicated in Figure V4. The disk file of symbolic execution results is described in Section V4.2.2.3.

```
COMMAND ACCEPTED:  XXXXXXXX
    INTERNAL MODEL USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE CONSTRAINT:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE INITIALIZED:  XXXXXXXX = XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    BRANCH OUTCOME SPECIFIED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    NUMBER OF ITERATIONS OF LOOP AT XXXX:  XXXX
COMMAND ACCEPTED:  XXXXXXXX
    LOCATION AT WHICH TO DISPLAY RESULTS:  XXXX
    RESULTS TO BE DISPLAYED:  XXXXXXXX

VALUE REQUIRED FOR UNRESOLVED ARRAY INDEX XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    ARRAY INDEX XXXXXXXX = XXXX
PATH COMPLETED

COMMAND ACCEPTED:  XXXXXXXX
        PATH DESCRIPTION:  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                                   .
                                   .
                                   .
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX
        PATH CONDITION:    XXXXXXXXXX &
                           XXXXXXXXXX &
                                .
                                .
                                .
                           XXXXXXXXXX
        SYMBOLIC VALUES:   XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                           XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                                   .
                                   .
                                   .
```

Figure V4. Format of the Symbolic Execution Interactive Dialogue

V4.2.2.3 Data Base: The program shall make use of the internal model stored on disk by the Preprocessor program. This model is described in Section V4.2.1.3. In addition, the Symbolic Executor shall store on disk the final symbolic execution results for each fully executed path. The format of this file shall be determined during detailed design of the system. The size of the file will depend upon the size of the SMITE program and the number of paths symbolically executed. This file is to be retained as long as the symbolic execution results may be of interest to the user.

APPENDIX W
FUNCTIONAL DESCRIPTION FOR THE TEST CASE GENERATOR

W1. GENERAL

W1.1 Purpose of Functional Description

See Appendix G.

W1.2 Project References

See Appendix G.

W2. SYSTEM SUMMARY

W2.1 Background

See Appendix G.

W2.2 Objectives

The purpose of the Test Case Generator is to aid in the selection of input values to be used for SMITE program testing. The system will interact with the user to identify input values that will cause user-selected program paths (sequences of source code statements) to be executed. These values can then be used as test cases for the SMITE program.

W2.3 Existing Methods and Procedures

Microprogram development at RADC is currently performed without the aid of a Test Case Generator. Identification of input values to execute specific paths of a SMITE program is a manual process.

W2.4 Proposed Methods and Procedures

The proposed method of obtaining test data for a SMITE program is to submit the program to the Test Case Generator. The Test Case Generator will operate in three phases, each of which may be executed independently.

Phase 1 will analyze the SMITE source code, generate a numbered source listing, and build an internal model of the program. Using this internal model together with inputs supplied interactively by the user, Phase 2 will symbolically execute the program, display the interactive dialogue, and generate a symbolic execution file. For each path explored during Phase 2 the symbolic execution file will contain a path description, identification of the input conditions that cause the path to be executed, symbolic values for all simple (i.e., non-array) variables, and a table identifying array references encountered on the path. The user may also request that these results be displayed as part of the interactive dialogue.

Phase 3 will use the symbolic execution file together with inputs supplied interactively by the user to identify input values that will cause user-selected paths to be executed. Phase 3 will display the interactive dialogue and will generate a test data file containing a set of input values for each path for which test data were identified.

Figure W1 illustrates the data flow for the system. Figure W2 shows its major processing steps.

W2.4.1 Summary of Improvements

The Test Case Generator will assist the user in identifying input values that will cause the execution of user-specified paths in a SMITE program. Specific benefits to be gained from this system are as follows:

- o The system will allow the user to observe the flow of control and flow of data during program execution.
- o The interactive capability of the system will permit the user to guide the symbolic execution based on intermediate results supplied by the system.
- o The system will be able to identify input values for some paths without assistance from the user.
- o When user assistance is necessary, the interactive dialogue will aid the user in directing further investigations.
- o For some paths, the system will be able to show that no solution is possible.
- o The length of time necessary for the user to identify input values for desired paths will be greatly reduced.

W2.4.2 Summary of Impacts

See Appendix G.

W2.5 <u>Expected Limitations</u>

To limit execution time and core requirements, the Test Case Generator will be designed to handle a SMITE program that is syntactically correct and that contains no more than 1,000 statements, 1,000 variables, 250 branches, and 10,000 variable occurrences.

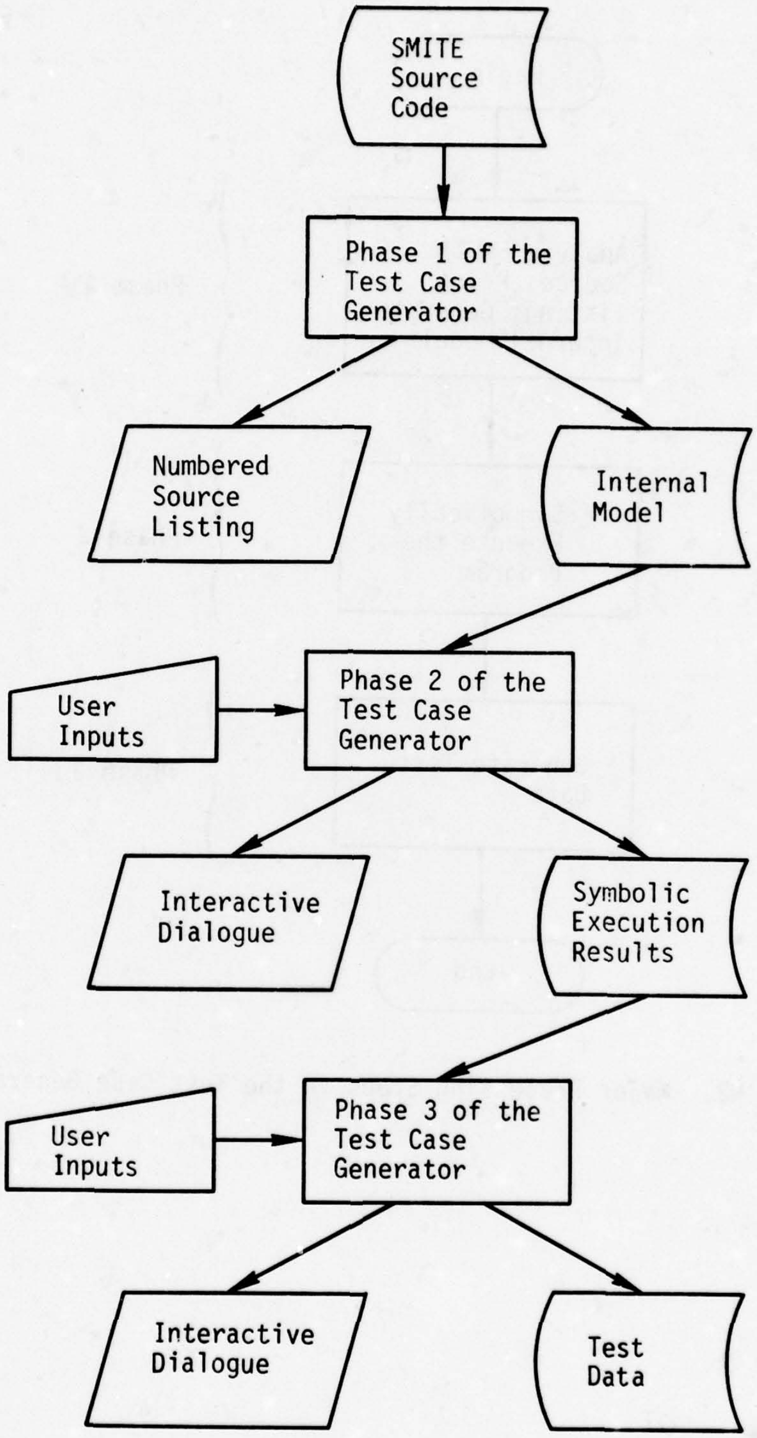

SMITE Source Code
Phase 1 of the Test Case Generator
Numbered Source Listing
Internal Model
User Inputs
Phase 2 of the Test Case Generator
Interactive Dialogue
Symbolic Execution Results
User Inputs
Phase 3 of the Test Case Generator
Interactive Dialogue
Test Data

Figure W1. Data Flow for the Test Case Generator

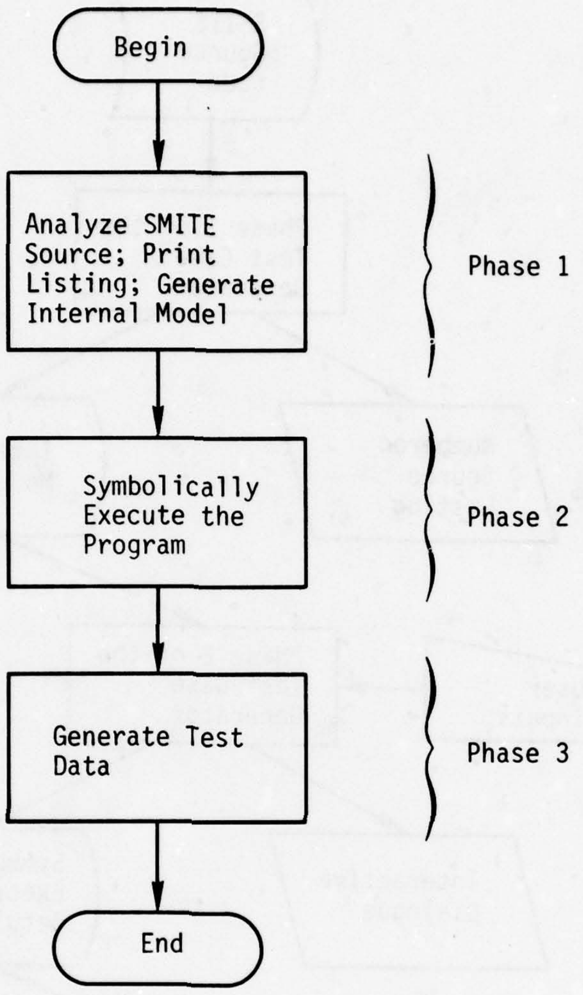

Begin
Analyze SMITE Source; Print Listing; Generate Internal Model
Phase 1
Symbolically Execute the Program
Phase 2
Generate Test Data
Phase 3
End

Figure W2. Major Processing Steps in the Test Case Generator

W3. DETAILED CHARACTERISTICS

W3.1 Specific Performance Requirements

The Test Case Generator shall operate in three phases, each designed to be executed independently. Phase 1 shall be responsible for performing the following tasks:

- o Analyzing the source code of a program written in SMITE
- o Assigning each source statement a sequence number and generating a numbered listing of the program
- o Building an internal model of the program

If any source statement cannot be interpreted during Phase 1, it shall be identified in the numbered listing and the system shall proceed to the next source statement. If any of the numerical limitations given in Section W2.5 are exceeded during the processing, the system shall report this problem in the listing and shall continue to process and list the source statements, storing all relevant information for which space remains. The internal model shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of each statement's function. The internal model shall be stored on disk to be used as input to Phase 2 of the system.

In Phase 2, the system shall use the internal model to execute the program symbolically. User inputs to this phase shall specify the internal model to be used, variable constraints and initializations, path selection commands, and the symbolic execution results that are to be displayed. Default options shall be provided. The system shall permit the user to submit a complete or partial set of inputs before symbolic execution begins. If a partial set of inputs is supplied, the system shall request additional inputs as needed. When such a request is made, the user shall also be permitted to insert other path selection commands, output requests, and processing specifications.

To symbolically execute the program, the system shall proceed along a path that is within the constraints specified by the user. When it encounters a statement that alters the value of a simple variable, it shall use substitution to update the internal representation of the variable. This substitution shall consist of taking the current symbolic value of each variable in the statement and using this value to replace the variable name. The resulting expression shall be the new symbolic value of the variable being modified. A table of array references shall be maintained for the path.

During processing, if the system encounters a branch it cannot resolve, it shall request further information, permit the user to respond interactively, and proceed. When the system reaches a point at which intermediate results have been requested, it shall output the requested results.

In executing the program symbolically, the system shall process through routine boundaries and shall correctly handle variable overlays.

Execution shall continue in this manner until the end of the path is reached. The system shall then, if requested, display on a terminal the results for that path, which may include:

- o The path description, identifying the order of the source code statements encountered in executing the paths
- o The path condition, i.e., the set of input conditions causing the path to be executed
- o Symbolic values for simple (i.e., non-array) variables
- o The array instance table, identifying array references encountered on the path

These results shall also be stored in a disk file for later output to a line printer or terminal. In the disk file, the results for each path shall be assigned an identification number. After directing the results to a disk file, the system shall permit the user to select one of the following processing options:

- o Process another path, if any, which satisfies the constraints
- o Cease execution of this class of paths and accept new input data
- o Terminate execution of Phase 2

Phase 3 shall use the disk file of symbolic execution results together with interactive input from the user to generate test data. The user inputs shall:

- o Designate the disk file of symbolic execution results to be used
- o Identify a path for which test data is desired, using the number assigned to the path in Phase 2
- o Assign numeric values to any variables necessary to resolve array references
- o Assign actual values to selected variables in the path condition

Phase 3 processing shall be organized into four successive steps:

- Allow the user to designate an input file
- Allow the user to designate a path
- Assist the user in resolving any unresolved array references
- Assist the user in solving the path condition

The user may return at any time to an earlier step, make changes to the inputs, and repeat this and all later steps. He may also terminate the execution of Phase 3 before all steps are completed. The steps are described below.

The user shall begin by specifying an input file and a path. Once these have been specified, the system shall determine whether there is an array instance table associated with the selected path. If so, the system shall request values for variables needed to resolve the array references. The system shall determine whether these values lead to a contradiction in the path condition. If so, the system shall interact with the user to modify these values until there is no contradiction in the path condition.

The system shall next attempt to solve the path condition. If a set of test data that satisfy the path condition is found, it shall be displayed for the user and stored in a disk file. If the system cannot find a solution, it shall request assistance from the user. The user shall then be permitted to interact with the system to assign actual values to variables until the system either solves the path condition or determines that no solution is possible.

The interactive dialogue shall be displayed on a terminal. The disk file of results shall contain a set of test data for each path for which a solution was found. Each set of test data shall be identified by its path identification number. The disk file shall be saved for later output to a line printer or terminal or for use as input when actually testing the SMITE program.

W3.1.1 Accuracy and validity

Accuracy and validity requirements are not applicable to the system.

W3.1.2 Timing

There are no timing requirements on the system.

W3.2 System Functions

The Test Case Generator shall perform three major functions, each of which shall be implemented as a separate program in the system. The functions shall be as follows:

- Preprocessing
- Symbolic Execution
- Test Data Generation

Preprocessing shall fulfill the requirements described in Section W3.1 for Phase 1 of the system. Symbolic Execution shall fulfill the requirements for Phase 2. Test Data Generation shall fulfill the requirements for Phase 3.

W3.3 Inputs/Outputs

W3.3.1 Inputs

The inputs to the Test Case Generator shall consist of:

- o The source code for a program written in SMITE
- o User-supplied inputs for Phase 2, which shall include:
  - Designation of the internal model to be used
  - Variable initializations and/or variable constraints in terms of numeric or symbolic values
  - Path selection commands, indicating the outcome of specified branches and the maximum number of iterations of specified loops
  - Output requests, specifying program locations at which symbolic execution results are to be displayed and the results that are to be displayed at each location
- o User-supplied inputs for Phase 3, which shall:
  - Designate the symbolic execution file to be used
  - Designate the path for which test data are desired
  - Assign numerical values to variables necessary to resolve array references
  - Assign actual values to user-specified variables in the path condition

The source code may be stored on punched cards, magnetic tape, or disk. Each logical record shall consist of an 80-character image. All other inputs shall be entered interactively via terminal.

W3.3.2 Outputs

The outputs of the system shall consist of:

- o A numbered listing of the SMITE source code

- o Interactive dialogue generated during Phase 2, which shall include intermediate and final results for each path as requested; these results shall include one or more of the following:
  - A path description, identifying the sequence of source statements encountered on the path
  - A path condition, describing the input conditions that caused the path to be executed
  - Symbolic values for selected simple variables
  - An array instance table identifying array references encountered on the path
- o Interactive dialogue generated during Phase 3, which shall include the test data values identified for each path
- o A disk file containing the test data that has been generated

The format of the numbered listing shall be as indicated in Figure W3. The formats of the interactive dialogue for Phases 2 and 3 shall be as indicated in Figures W4 and W5, respectively.

W3.4 Data Characteristics

The SMITE program to be processed may reside in a system data base. In addition, the system shall create the following files to be stored on disk:

- o A file containing the internal model of the program
- o A file of symbolic execution results, which shall contain for each explored path:
  - A path identification number
  - A path description
  - A path condition
  - Symbolic values for all simple variables
  - An array instance table
- o A file which shall contain for each path for which test data was found:
  - A path identification number
  - A set of test data that will cause the path to be executed

The format of each file shall be determined during detailed design of the system. The size of the internal model will depend upon the SMITE program

```
STATEMENT #            SOURCE STATEMENT

 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .                    .
   .                    .
   .                    .
```

Figure W3. Format of the Numbered Source Listing

```
COMMAND ACCEPTED:  XXXXXXXX
    INTERNAL MODEL USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE CONSTRAINT:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    VARIABLE INITIALIZED:  XXXXXXXX = XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    BRANCH OUTCOME SPECIFIED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    NUMBER OF ITERATIONS OF LOOP AT XXXX:  XXXX
COMMAND ACCEPTED:  XXXXXXXX
    LOCATION AT WHICH TO DISPLAY RESULTS:  XXXX
    RESULTS TO BE DISPLAYED:  XXXXXXXX
PATH NUMBER XXXX COMPLETED

COMMAND ACCEPTED:  XXXXXXXX
        PATH DESCRIPTION:  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                                  .
                                  .
                           XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

        PATH CONDITION:    XXXXXXXXXX &
                           XXXXXXXXXX &
                                .
                                .
                                .
                           XXXXXXXXXX

        SYMBOLIC VALUES:   XXXXXXXX = XXXXXXXXXXXXXXXXXXXXX
                           XXXXXXXX = XXXXXXXXXXXXXXXXXXXXX
                                  .
                                  .
                                  .

        ARRAY INSTANCES:
                             ARRAY          INDICES            SYMBOLIC VALUE

                           XXXXXXXX    XXXXXXXX  XXXXXXXX    XXXXXXXXXXXXXXXX
                           XXXXXXXX    XXXXXXXX  XXXXXXXX    XXXXXXXXXXXXXXXX
                              .            .         .              .
                              .            .         .              .
                              .            .         .              .
```

Figure W4. Format of the Symbolic Execution Interactive Dialogue

```
COMMAND ACCEPTED:  XXXXXXXX
    SYMBOLIC EXECUTION RESULTS USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    PATH NUMBER XXXX SELECTED
COMMAND ACCEPTED:  XXXXXXXX
    VALUE XXXXXXXX ASSIGNED TO VARIABLE XXXXXXXX
    ARRAY INSTANCES RESOLVED
COMMAND ACCEPTED:  XXXXXXXX
    VALUE XXXX ASSIGNED TO VARIABLE XXXXXXXX


    DATA VALUES FOR PATH XXXX:

        XXXXXXXX = XXXX
        XXXXXXXX = XXXX
                 .
                 .
                 .
```

Figure W5. Format of the Test Data Generation Interactive Dialogue

being processed, but should not exceed 50K words. This file is to be retained as long as it may be needed for execution of Phase 2 of the system. The size of the symbolic execution results file will depend upon the SMITE program being analyzed and the number of paths traversed. This file is to be retained as long as it may be needed for execution of Phase 3 of the system. The size of the test data file will depend upon the number of paths analyzed and the number of test data values required for each path. This file is to be retained as long as the test data it contains is of value in testing the SMITE program.

W3.5 Failure Contingencies

Not applicable.

W4. ENVIRONMENT

W4.1 Equipment Environment

The Test Case Generator will operate on RADC's DEC System 20, which is tied to the ARPANET. It will require approximately 250K words of main memory to operate. The system will be stored on disk. The SMITE source code will be input via card reader, magnetic tape, or disk. All other inputs will be entered via terminal. The internal model, the file of symbolic execution results, and the file of test data generation results will be stored on disk. All other output will be directed to a terminal or line printer. No new equipment will be required.

W4.2 Support Software Environment

See Appendix G.

W4.3 Interfaces

The system will not interface with any other system.

W4.4 Security

The system will have no classified components.

W5. COST FACTORS

Development of the Test Case Generator will require approximately 60 man-months of effort. Continuing cost factors will be limited to the costs incurred in the use and maintenance of the program.

W6. DEVELOPMENTAL PLAN

The Test Case Generator is one of the microprogramming tools recommended by Logicon during the Reliable Microprogramming Contract. RADC will select from among these tools those that are to be implemented. The overall development schedule will depend upon the tools selected. The recommended schedule for development of the Test Case Generator is given in Figure W6.

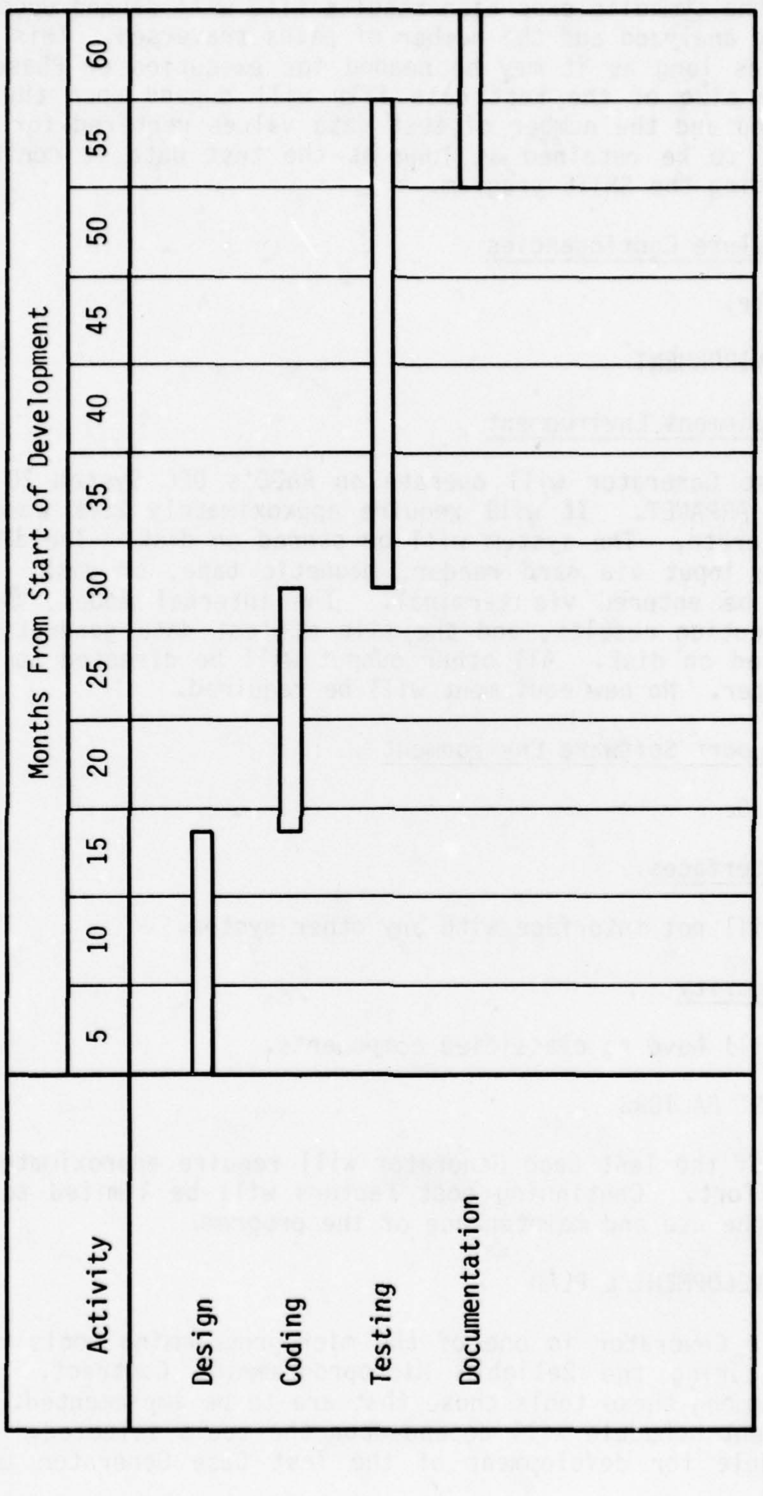


Figure W6. Development Schedule for the Test Case Generator

APPENDIX X
SYSTEM SPECIFICATION FOR THE TEST CASE GENERATOR

X1. GENERAL

X1.1 Purpose of the System Specification

See Appendix H.

X1.2 Project References

See Appendix H.

X2. SUMMARY OF REQUIREMENTS

X2.1 System Description

The Test Case Generator shall assist the user in the selection of input values to be used for SMITE program testing. The system shall interact with the user to identify input values that will cause user-selected program paths (i.e., sequences of source code statements) to be executed.

The system shall operate in three independent phases. Phase 1 shall analyze the SMITE source code, generate a numbered source listing, and build an internal model of the program. Phase 2 shall use the internal model together with inputs supplied interactively by the user to execute the program symbolically and generate results consisting of a description of each path, the input conditions that caused the path to be executed, symbolic values for all simple (i.e., non-array) variables, and a table identifying array references encountered on the path. Phase 3 shall use the symbolic execution results together with inputs supplied interactively by the user to identify test data that will cause user-specified paths to be executed.

The system shall consist of three computer programs. Figure X1 identifies these programs, illustrates their relationship to each other, and shows the phase to which each belongs.

X2.2 System Functions

The Test Case Generator shall be comprised of the following functions, each of which shall be implemented as a separate program in the system:

- o Preprocessing
- o Symbolic Evaluation
- o Test Data Generation

The following paragraphs identify the requirements to be satisfied by each of these functions.

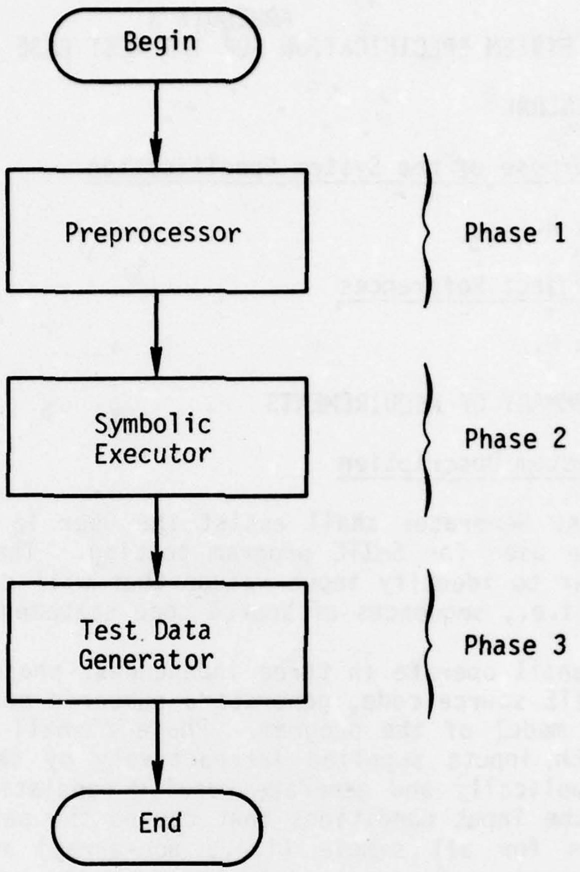


Figure X1. Relationship of the Components of the Test Case Generator

X2.2.1 Preprocessing Function

The Preprocessing function shall accept as input the source code of a program written in SMITE. The function shall process each source statement of the program as follows:

- o Assign the statement a sequence number
- o Output a copy of the statement and its sequence number in a listing of the program
- o Analyze the contents of the statement and use it in building an internal model of the program

Source code comments shall be copied into the program listing without being analyzed or assigned a sequence number. The internal model shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of each statement's function.

To limit execution time and core requirements, the function shall expect the program to be syntactically correct and to have no more than 1,000 statements, 1,000 variables, 250 branches, and 10,000 variable occurrences. If any source statement cannot be interpreted, the function shall report this problem in the program listing and shall proceed to the next source statement. If any of the numerical limitations given above is exceeded during the processing, the function shall report this problem in the listing, then continue to analyze and list each source statement, storing all relevant information for which space remains.

At the conclusion of preprocessing, the function shall output a problem summary message if any source statements could not be interpreted or if space limitations were exceeded during the processing. The internal model shall be stored on disk to be used as input to the Symbolic Execution function.

X2.2.2 Symbolic Execution Function

The Symbolic Execution function shall use the internal model produced by the Preprocessing function to execute the SMITE program symbolically. User inputs to be supplied before symbolic execution begins shall be as follows:

- o Designation of the internal model to be used
- o Variable constraints and initializations
- o Path selection commands indicating the outcome of specified branches and the maximum number of iterations of specified loops

- o Output requests specifying the program locations at which symbolic execution results are to be displayed and the results that are to be displayed at each location

Provision for defaults shall be as follows:

- o There shall be no default for the internal model to be used.
- o Variable constraints and initializations:
  - No variable constraints shall be defined by default.
  - The symbolic value of each variable shall be undefined unless the user explicitly assigns a numeric or symbolic value. For all remaining undefined variables, the user shall have the option of specifying with a single command that the symbolic value for each is to be the name of the variable.
- o Path selection commands:
  - The function shall request user input at each branch point whose outcome has not been previously specified.
  - All unspecified loops shall be executed once.
- o Intermediate and final results for each path shall not be displayed unless requested by the user.

The function shall permit the user to submit a complete or partial set of inputs before symbolic execution begins. If a partial set of inputs is supplied, the function shall request additional inputs as needed. When such a request is made, the user shall also be permitted to insert other path selection commands, output requests, requests that the function cease execution of this class of paths and accept new input data, or requests that the function cease execution altogether.

To execute the program symbolically, the function shall proceed along a path that is within the constraints specified by the user. When it encounters a statement that modifies the value of a simple variable, it shall use substitution to update the internal representation of the variable. This substitution shall consist of taking the current symbolic value of each variable in the statement and using this value to replace the variable name. The resulting expression shall be the new symbolic value of the variable being modified. A table of array references shall be maintained for the path. The function shall process through routine boundaries and shall correctly handle variable overlays.

If the function encounters an undefined variable, it shall permit the user either to initialize the variable or to leave the variable undefined. If the variable is left undefined, the function shall consider the current path to be terminated.

When the function reaches a point at which intermediate results have been requested, it shall display the requested results, permit user input, then proceed. The intermediate results may include:

- o A path description: a sequence of statement numbers identifying the order of the source code statements encountered in executing the path
- o A path condition: the set of input conditions causing the path to be executed
- o Symbolic values for simple (i.e., non-array) variables
- o An array instance table identifying array references encountered on the path

If the function encounters a branch point it cannot resolve, it shall permit the user to specify one or more branches to be executed. If the user selects more than one branch, the system shall note the order in which these branches were selected, preserve the path condition to this branch point, and continue processing, using the first branch selection.

When the function completes a path, it shall, if requested, display the final results for that path. These results shall be as described above for the intermediate results. Unless the path has been terminated prematurely because of an undefined variable, these results shall be stored on disk for later output to a terminal or line printer and for use as input to the Test Data Generation function. As each set of results is stored on disk, it shall be assigned a path identification number. The function shall then permit the user to select one of the following processing options:

- o Process another path which satisfies user constraints, if there are any
- o Cease execution of this class of paths and accept new input data
- o Terminate symbolic execution

If the user selects the first option, the system shall return to the most recent branch point for which there is a selected branch yet to be executed. If such a branch point exists, the function shall request inputs specifying whether the user wishes to proceed with the next branch selection. If the user's reply is affirmative, the function shall execute this branch. If the reply is negative, the function shall offer another branch selection. When no branch point exists at which a selected branch remains to be executed, the function shall cease execution of this class of paths. The user shall then be permitted to submit a new set of inputs or to terminate symbolic execution.

X2.2.3 Test Data Generation Function

The Test Data Generation function shall use the file of symbolic execution results produced by the Symbolic Execution function, together with interactive inputs from the user, to generate test data. The user inputs shall:

- o Designate the disk file of symbolic execution results to be used
- o Identify a path for which test data are desired
- o Assign numeric values to variables necessary to resolve array references
- o Assign actual values to selected variables in the path condition

The function shall be organized into four successive steps. These steps shall:

- o Allow the user to designate an input file
- o Allow the user to designate a path
- o Assist the user in resolving any unresolved array references
- o Assist the user in identifying values that satisfy the path condition

The user may return at any time to an earlier step, make changes to the inputs, and repeat this and all later steps. He may also terminate execution of the function before all steps are completed. The steps are described below.

The user may return at any time to an earlier step, make changes to the inputs, and repeat this and all later steps. He may also terminate execution of the function before all steps are completed. The steps are described below.

The user shall begin by specifying an input file and a path. Once these have been specified, the function shall determine whether there is an array instance table associated with the selected path. If so, the function shall request values for variables needed to resolve the array references. The function shall determine whether these values lead to a contradiction in the path condition. If so, the function shall interact with the user until values are provided that cause no contradiction in the path condition.

The function shall next attempt to solve the path condition. If a set of values that satisfy the path condition is found, the values shall be displayed for the user and stored in a disk file. If the function cannot find a solution, it shall request assistance from the user. The user shall then be permitted to interact with the system by assigning actual values to variables. The function shall verify that the values provided do not contradict values assigned while resolving the array references, and shall use the values to attempt to solve the path condition. This interactive process shall continue until the function solves the path condition, determines that no solution is possible, or is directed to return to an earlier step to receive different inputs.

The interactive dialogue shall be displayed on a terminal. The disk file of results shall contain a set of test data for each path for which a solution was found. Each set of test data shall be identified by its path identification number. The disk file shall be saved for later output to a line printer or terminal or for use as input when actually testing the SMITE program.

X2.3 Accuracy and Validity

Accuracy and validity requirements are not applicable to the system.

X2.4 Timing

There are no timing requirements on the system.

X2.5 Flexibility

The system shall be designed in such a way that the limit on the number of statements, variables, branches, and variable occurrences can be easily modified.

X3. ENVIRONMENT

X3.1 Equipment Environment

See Appendix H for equipment description.

The Test Case Generator will require approximately 250K words of main memory to operate. The system will be stored on disk, requiring approximately 90K words of storage for the source code, object code, and load module. The user-prepared inputs shall be entered via terminal. The SMITE source code may be input from punched cards, magnetic tape, or disk. The internal model, the file of symbolic execution results, and the file of test data shall be stored on disk. The other outputs may be directed to a line printer or to a terminal. No new equipment will be required.

X3.2 Support Software Environment

See Appendix H.

X3.3 Interfaces

The system shall not interface with any other system.

X3.4 Security

The system shall have no classified components.

X3.5 Controls

No controls shall be established by the system.

X4. DESIGN DATA

X4.1 System Logical Flow

The Test Case Generator shall be a system consisting of three computer programs:

- o Preprocessor
- o Symbolic Executor
- o *Test Data Generator*

Each of these programs shall constitute one phase of the system. For a given SMITE program version, the Symbolic Executor can be executed any number of times once the Preprocessor has been executed at least once, and the Test Data Generator can be executed any number of times for each execution of the Symbolic Executor. Figure X2 indicates a typical logical flow for the system and shows the system's inputs and outputs, including the data base files created and used by each program.

X4.2 Program Descriptions

The Test Case Generator shall be comprised of the three computer programs identified in Section X4.1. Design data for each program follow.

X4.2.1 Preprocessor Program

The Preprocessor shall fulfill the requirements specified in Section X2.2.1. Additional design data follow.

X4.2.1.1 Inputs: The input to the program shall consist of the source code for a program written in SMITE. The logical records in the file shall be 80-character images of the format specified in the SMITE Training Manual (RADC-TR-77-364, 12 August 1977).

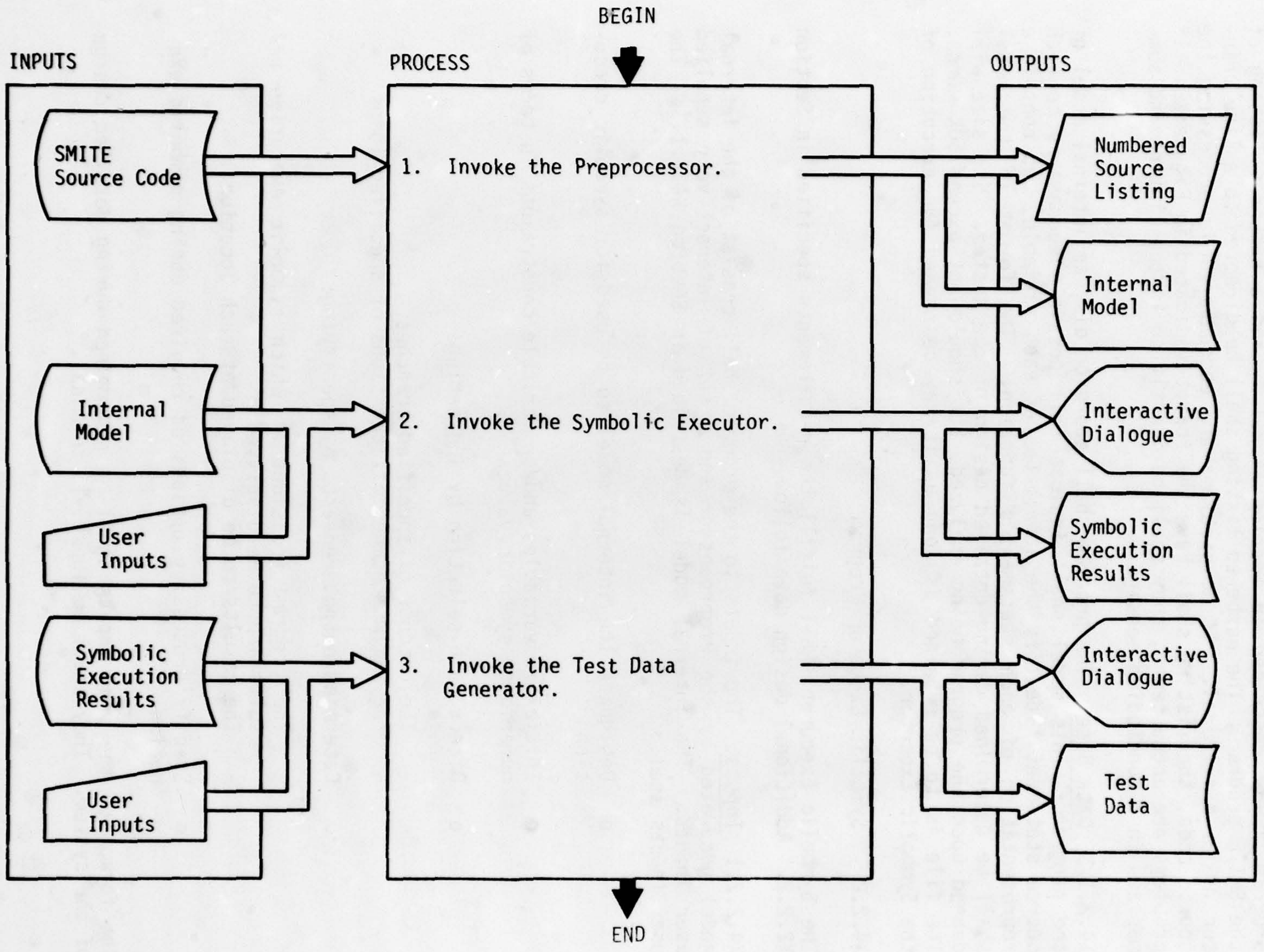


Figure X2. System Logical Flow for the Test Case Generator

X4.2.1.2 <u>Outputs</u>: The output of the program shall consist of an internal program model, described in Section X4.2.1.3, and a numbered listing of the SMITE program. The numbered listing shall be directed to a line printer or to a terminal. If no problems are encountered in processing the source code, the listing shall have the format indicated in Figure X3. If problems are encountered, they shall be identified in the listing and summarized in a concluding message.

X4.2.1.3 Data Base: The program shall store on disk an internal model of the program. This model shall indicate all possible successors to each source statement, identify the attributes of each variable, and contain a representation of each statement's function. The format of the model shall be determined during detailed design of the system. Its size will depend upon the program being analyzed, but should not exceed 50K words. The file is to be retained as long as it may be needed for execution of the Symbolic Executor.

X4.2.2 Symbolic Executor Program

The Symbolic Executor shall fulfill the requirements specified in Section X2.2.2. Additional design data follow.

X4.2.2.1 Inputs: The inputs to the program shall consist of the internal model generated by the Preprocessor and a set of interactively supplied user inputs. The internal model is described in Section X4.2.1.3. The user inputs shall:

- o Designate the internal model to be used for symbolic execution
- o Initialize variables and/or variable constraints in terms of numeric or symbolic values
- o Direct path selection by indicating:
  - The outcome of specified branches
  - The maximum number of iterations of specified loops
- o Determine output requests by specifying:
  - The program locations at which symbolic execution results are to be displayed
  - The results to be displayed at each location
- o Specify processing options as required during symbolic execution

The format of the user inputs shall be determined during detailed design of the system. They shall be input via terminal.

```
STATEMENT #             SOURCE STATEMENT

 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXX      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   .
   .
   .
```

Figure X3. Format of the Numbered Source Listing

X4.2.2.2 Outputs: The output of the program shall consist of interactive dialogue between the user and the system, including symbolic execution results for each path explored. The program may initiate interactive dialogue by requesting the user to specify:

- o The internal model to be used
- o Values for undefined variables
- o Branch selections
- o The symbolic execution results to be displayed at intermediate and final points on a path
- o The processing to be performed upon completion of a path

The symbolic execution results for each path shall consist of the following:

- o A path description, identifying the order of the source statements encountered during execution of the path
- o A path condition, consisting of the set of input conditions that caused the path to be executed
- o Symbolic values of simple (i.e., non-array) variables, expressed in terms of the input variables
- o An array instance table identifying array references encountered on the path

The interactive dialogue shall be displayed on a terminal. It shall include intermediate and final symbolic execution results for each path as requested. The final symbolic execution results for each fully executed path shall be directed to a disk file for later output to a terminal or line printer and for use as input to the Test Data Generator. The format of the interactive dialogue and symbolic execution results that are displayed on the terminal shall be as indicated in Figure X4. The disk file of symbolic execution results is described in Section X4.2.2.3.

X4.2.2.3 Data Base: The program shall make use of the internal model stored on disk by the Preprocessor program. This model is described in Section X4.2.1.3. In addition, the Symbolic Executor shall store on disk the final symbolic execution results for each fully executed path. These results shall consist of a path identification number, a path description, a path condition, symbolic values for all simple variables, and an array instance table. The format of this file shall be determined during detailed design of the system. Its size will depend upon the size of the SMITE program and the number of paths symbolically executed. It is to be retained as long as it may be needed for executions of the Test Data Generator.

```
COMMAND ACCEPTED:  XXXXXXXX
   INTERNAL MODEL USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
   VARIABLE CONSTRAINT:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
   VARIABLE INITIALIZED:  XXXXXXXX = XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
   BRANCH OUTCOME SPECIFIED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
   NUMBER OF ITERATIONS OF LOOP AT XXXX:  XXXX
COMMAND ACCEPTED:  XXXXXXXX
   LOCATION AT WHICH TO DISPLAY RESULTS:  XXXX
   RESULTS TO BE DISPLAYED:  XXXXXXXX
PATH NUMBER XXXX COMPLETED

COMMAND ACCEPTED:  XXXXXXXX
      PATH DESCRIPTION:  XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                         XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-
                           .
                           .
                         XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX

      PATH CONDITION:    XXXXXXXXXX &
                         XXXXXXXXXX &
                           .
                           .
                         XXXXXXXXXX

      SYMBOLIC VALUES:   XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                         XXXXXXXX = XXXXXXXXXXXXXXXXXXXX
                           .
                           .

      ARRAY INSTANCES:

                           ARRAY          INDICES              SYMBOLIC VALUE

                         XXXXXXXX    XXXXXXXX  XXXXXXXX     XXXXXXXXXXXXXXXX
                         XXXXXXXX    XXXXXXXX  XXXXXXXX     XXXXXXXXXXXXXXXX
                            .           .         .               .
                            .           .         .               .
                            .           .         .               .
```

Figure X4. Format of the Symbolic Execution Interactive Dialogue

X4.2.3 Test Data Generator Program

The Test Data Generator shall fulfill the requirements specified in Section X2.2.3. Additional design data follow.

X4.2.3.1 Inputs: The inputs to the program shall consist of a file of symbolic execution results generated by the Symbolic Executor and a set of interactively supplied user inputs. The file of symbolic execution results is described in Section X4.2.2.3. The user inputs shall:

- Designate the disk file of symbolic results to be used
- Designate a path for which test data are desired
- Assign numeric values to any variables necessary to resolve array references
- Assign actual values to selected variables in the path condition

The format of the user inputs shall be determined during detailed design of the system. They shall be entered via terminal.

X4.2.3.2 Outputs: The output of the program shall consist of a file of test data, described in Section X4.2.3.3, and interactive dialogue between the user and the system. The program may initiate interactive dialogue by requesting the user to specify the inputs described in Section X4.2.3.1. The interactive dialogue shall be displayed on a terminal and shall be formatted as indicated in Figure X5.

X4.2.3.3 Data Base: The program shall make use of a file of symbolic execution results generated by the Symbolic Executor. This file is described in Section X4.2.2.3. In addition, the program shall store on disk a set of test data for each path for which a solution was found during execution of the Test Data Generator. Each set of test data shall be labeled with the appropriate path identification number. The format of the test data file shall be determined during detailed design of the system. The size of the file will depend upon the number of paths analyzed and the number of test data values required for each path. The file is to be retained as long as the test data it contains may be of use in SMITE program testing.

```
COMMAND ACCEPTED:  XXXXXXXX
    SYMBOLIC EXECUTION RESULTS USED:  XXXXXXXX
COMMAND ACCEPTED:  XXXXXXXX
    PATH NUMBER XXXX SELECTED
COMMAND ACCEPTED:  XXXXXXXX
    VALUE XXXXXXXX ASSIGNED TO VARIABLE XXXXXXXX
    ARRAY INSTANCES RESOLVED
COMMAND ACCEPTED:  XXXXXXXX
    VALUE XXXX ASSIGNED TO VARIABLE XXXXXXXX

    DATA VALUES FOR PATH XXXX:
        XXXXXXXX = XXXX
        XXXXXXXX = XXXX
           .
           .
           .
```

Figure X5. Format of the Test Data Generation Interactive Dialogue

☆U.S. GOVERNMENT PRINTING OFFICE: 1979-614-023/209